



INTRODUCCIÓN A LA PROGRAMACIÓN CON

PYTHON

Algoritmos
y lógica de programación
para principiantes

novatec

Nilo Ney Coutinho Menezes

INTRODUCCIÓN A LA PROGRAMACIÓN CON PYTHON

Algoritmos y lógica de programación para principiantes

Nilo Ney Coutinho Menezes

Novatec

Edición original en portugués del libro Introdução à Programação com Python 2ª edição, ISBN 9788575224083, publicada por Novatec Editora Ltda. © 2014 Novatec Editora Ltda.

Traducción al Español realizada por Novatec Editora. © 2016 Novatec Editora Ltda.

Todos los derechos reservados.

Esta traducción es impresa y vendida con el permiso de Novatec Editora, poseedora de todos los derechos para publicación y venta de esta obra. Está prohibida la reproducción de esta obra, aun parcial, por cualquier proceso, sin previa autorización, por escrito, del autor y de la Editorial.

Editor: Rubens Prates

Traducción al español: Ricardo Pérez Banega

Revisión gramatical: Pilar Domingo

Edición electrónica: Camila Kuwabata

Tapa: Victor Bittow

ISBN: 978-85-7522-519-6

Novatec Editora Ltda.

Rua Luís Antônio dos Santos 110

02460-000 – São Paulo, SP – Brasil

Tel.: +55 11 2959-6529

E-mail: novatec@novatec.com.br

Sitio: novatec.com.br

Twitter: twitter.com/novateceditora

Facebook: facebook.com/novatec

LinkedIn: linkedin.com/in/novatec

Edición en español impresa y distribuida en el exterior por:

Coutinho Menezes Nilo - LogiKraft

Rue de la Grande Campagne, 40

7340 Wasmes

Belgium

+32 485 251460

libros@logikraft.be

A mi esposa, Chris; y a mis hijos, Igor, Hanna e Iris.

Tabla de contenido

Capítulo 1 Motivación

- [1.1 ¿Ud. quiere aprender a programar?](#)
- [1.2 ¿Cuál es su nivel de paciencia?](#)
- [1.3 ¿Cuánto tiempo pretende estudiar?](#)
- [1.4 ¿Para qué programar?](#)
 - [1.4.1 Escribir páginas web](#)
 - [1.4.2 Poner en hora su reloj](#)
 - [1.4.3 Aprender a usar mapas](#)
 - [1.4.4 Mostrarle a sus amigos que sabe programar](#)
 - [1.4.5 Parecer extraño](#)
 - [1.4.6 Entender mejor cómo funciona su computadora](#)
 - [1.4.7 Cocinar](#)
 - [1.4.8 Salvar el mundo](#)
 - [1.4.9 Software libre](#)
- [1.5 ¿Por qué Python?](#)

Capítulo 2 Preparando el ambiente

- [2.1 Instalación de Python](#)
 - [2.1.1 Windows](#)
 - [2.1.2 Linux](#)
 - [2.1.3 Mac OS X](#)
- [2.2 Usando el intérprete](#)
- [2.3 Editando archivos](#)
- [2.4 Cuidados al digitar sus programas](#)
- [2.5 Los primeros programas](#)
- [2.6 Conceptos de variables y atribución](#)

Capítulo 3 Variables y entrada de datos

- [3.1 Nombres de las variables](#)
- [3.2 Variables numéricas](#)
 - [3.2.1 Representación de valores numéricos](#)
- [3.3 Variables del tipo lógico](#)
 - [3.3.1 Operadores de comparación](#)
 - [3.3.2 Operadores lógicos](#)
- [3.4 Variables del tipo de cadena de caracteres](#)
 - [3.4.1 Operaciones con cadenas de caracteres](#)
- [3.5 Secuencias y tiempo](#)
- [3.6 Rastreo](#)
- [3.7 Entrada de datos](#)
 - [3.7.1 Conversión de la entrada de datos](#)
 - [3.7.2 Errores comunes](#)

Capítulo 4 Condiciones

- [4.1 if](#)
- [4.2 else](#)
- [4.3 Estructuras anidadas](#)
- [4.4 elif](#)

Capítulo 5 Repeticiones

- [5.1 Contadores](#)

- [5.2 Acumuladores](#)
- [5.3 Interrumpiendo la repetición](#)
- [5.4 Repeticiones anidadas](#)

Capítulo 6 Listas

- [6.1 Trabajando con índices](#)
- [6.2 Copia y rebanado de listas](#)
- [6.3 Tamaño de listas](#)
- [6.4 Adición de elementos](#)
- [6.5 Remoción de elementos de la lista](#)
- [6.6 Usando listas como colas](#)
- [6.7 Uso de listas como pilas](#)
- [6.8 Investigación](#)
- [6.9 Usando for](#)
- [6.10 Range](#)
- [6.11 Enumerate](#)
- [6.12 Operaciones con listas](#)
- [6.13 Aplicaciones](#)
- [6.14 Listas con cadenas de caracteres](#)
- [6.15 Listas dentro de listas](#)
- [6.16 Ordenamiento](#)
- [6.17 Diccionarios](#)
- [6.18 Diccionarios con listas](#)
- [6.19 Tuplas](#)

Capítulo 7 Trabajando con cadenas de caracteres

- [7.1 Verificación parcial de cadenas de caracteres](#)
- [7.2 Recuento](#)
- [7.3 Investigación de cadenas de caracteres](#)
- [7.4 Posicionamiento de cadenas de caracteres](#)
- [7.5 Separación de cadenas de caracteres](#)
- [7.6 Sustitución de cadenas de caracteres](#)
- [7.7 Remoción de espacios en blanco](#)
- [7.8 Validación por tipo de contenido](#)
- [7.9 Formateo de cadenas de caracteres](#)
 - [7.9.1 Formateo de números](#)
- [7.10 Juego del ahorcado](#)

Capítulo 8 Funciones

- [8.1 Variables locales y globales](#)
- [8.2 Funciones recursivas](#)
- [8.3 Validación](#)
- [8.4 Parámetros opcionales](#)
- [8.5 Nombrando parámetros](#)
- [8.6 Funciones como parámetro](#)
- [8.7 Empaquetamiento y desempaquetamiento de parámetros](#)
- [8.8 Desempaquetamiento de parámetros](#)
- [8.9 Funciones Lambda](#)
- [8.10 Módulos](#)
- [8.11 Números aleatorios](#)
- [8.12 La función type](#)

Capítulo 9 Archivos

- [9.1 Parámetros de la línea de comando](#)
- [9.2 Generación de archivos](#)
- [9.3 Lectura y escritura](#)
- [9.4 Procesamiento de un archivo](#)
- [9.5 Generación de HTML](#)

- [9.6 Archivos y directorios](#)
- [9.7 Un poco sobre el tiempo](#)
- [9.8 Uso de directorios](#)
- [9.9 Visita a todos los subdirectorios recursivamente](#)

Capítulo 10 Clases y objetos

- [10.1 Objetos como representación del mundo real](#)
- [10.2 Pasaje de parámetros](#)
- [10.3 Ejemplo de un banco](#)
- [10.4 Herencia](#)
- [10.5 Desarrollando una clase para controlar listas](#)
- [10.6 Revisando la agenda](#)

Capítulo 11 Banco de datos

- [11.1 Conceptos básicos](#)
- [11.2 SQL](#)
- [11.3 Python & SQLite](#)
- [11.4 Consultando registros](#)
- [11.5 Actualizando registros](#)
- [11.6 Borrando registros](#)
- [11.7 Simplificando el acceso sin cursores](#)
- [11.8 Accediendo a los campos como en un diccionario](#)
- [11.9 Generando una clave primaria](#)
- [11.10 Alterando la tabla](#)
- [11.11 Agrupando datos](#)
- [11.12 Trabajando con fechas](#)
- [11.13 Claves y relaciones](#)
- [11.14 Convirtiendo la agenda para utilizar un banco de datos](#)

Capítulo 12 Próximos pasos

- [12.1 Programación funcional](#)
- [12.2 Algoritmos](#)
- [12.3 Juegos](#)
- [12.4 Orientación a objetos](#)
- [12.5 Banco de datos](#)
- [12.6 Sistemas web](#)
- [12.7 Otras bibliotecas Python](#)
- [12.8 Listas de discusión](#)

Apéndice A Mensajes de error

- [A.1 SyntaxError](#)
- [A.2 IndentationError](#)
- [A.3 KeyError](#)
- [A.4 NameError](#)
- [A.5 ValueError](#)
- [A.6 TypeError](#)
- [A.7 IndexError](#)

Agradecimientos

Este libro no sería posible sin el incentivo de mi esposa, Emília Christiane, y la comprensión de mis hijos, Igor, Hanna e Iris. Para quien vive en un país frío como Bélgica, escribir en el verano no siempre es fácil.

También agradezco el apoyo ofrecido por mis padres y abuelos durante mis estudios, y los consejos, comprensión y orientación que siempre recibí.

No podría olvidarme de mi hermano, Luís Victor, por su ayuda con las imágenes en español.

A Luciano Ramalho y a los colegas de la lista python-brasil. Luciano, gracias por el incentivo para publicar este libro y por los comentarios más que pertinentes. A la comunidad python-brasil por el esfuerzo y prueba de civilidad al mantener las discusiones en altísimo nivel, ayudando a principiantes, curiosos y profesionales de la computación.

Agradezco también a los colegas, amigos y alumnos de la Fundación Matias Machline, hoy Fundación Nokia de Enseñanza, donde tuve oportunidad de estudiar y trabajar como profesor de lógica de programación. A los amigos y colegas del Centro Educacional La Salle y de la Fundación Paulo Feitoza, donde dicté cursos de lógica de programación y de Python.

Prefacio de la edición española

Desde que escribí la primera edición de este libro en 2010, siempre quise de traducirlo a otras lenguas. Como trabajé por un período corto, pero rico, con instituciones en la República Dominicana, el deseo de producir una versión en español del libro quedó muy claro. Todo esfuerzo de traducción y revisión fue realizado de manera de obtener un texto técnico en español que pueda ser utilizado tanto en España como en América Latina. Usted verá que algunos ejemplos del libro se refieren a Brasil, mi país natal, pero que todo el contenido, inclusive los comentarios de cada programa, nombres de variables y de funciones fueron traducidos al español. El sitio web del libro en español también es independiente de la versión en portugués de manera de posibilitar una experiencia auténtica en español. Si aún así, algún término o explicación no están claros o si encuentra palabras o fragmentos que no correspondan a su variedad local del español, no dude en enviar un email a errores@librodepython.com.

Este libro obtuvo una excelente aceptación en la comunidad brasileña de informática, donde fue adoptado en cursos introductorios de programación de computadoras en diversas facultades, y también por personas que decidieron aprender a programar solas, solo con la ayuda de este libro. Espero que la versión en español tenga también una gran aceptación y que la comunidad participe en la elaboración de nuevas ediciones, enviando sus dudas y sugerencias.

Prefacio de la segunda edición

Revisar este libro fue un trabajo muy gratificante, especialmente por los mensajes de apoyo recibidos a través de la lista Python-Brasil y por e-mail. Esta nueva edición trae un capítulo extra, cubriendo lo básico de banco de datos, donde introducimos el uso del SQLite. El capítulo 10, que trata de la orientación a objetos, también fue expandido, y se agregaron varios conceptos específicos de Python. Muchas pequeñas revisiones fueron hechas por todo el libro, desde la actualización para Python 3.4 hasta la migración para Windows 8.1.

A pesar de la tentación de cubrir todo lo que se pueda imaginar sobre Python, este libro continúa siendo una obra dedicada a aquellos que dan sus primeros pasos en la programación de computadoras. En esta nueva edición, conceptos más específicos de Python también fueron abordados para hacer el texto y nuestros programas más “Pythonicos”.

Prefacio de la primera edición

Aprendí a programar usando el lenguaje BASIC a mediados de los años 1980. Recuerdo construir pequeños programas de dibujo, agendas telefónicas y juegos. Almacenamiento sólo en cintas K-7. Antes de la internet, y viviendo en el norte del Brasil, la forma de aprender a programar era leyendo libros y, claro, programando. La forma más común de obtener nuevos programas era por medio de revistas de programación. Esas revistas estaban verdaderamente dedicadas a la nueva comunidad de usuarios de microcomputadores, término usado en la época para diferenciar las computadoras domésticas. Traían listas completas de programas escritos en BASIC o *Assembly*. En una época en que el download era un sueño distante, digitar esos programas era la única solución para poder ejecutarlos. La experiencia de leer y digitar los programas fue muy importante para aprender a programar pero hoy, desgraciadamente, pocas revistas técnicas son accesibles para los principiantes. La complejidad de los programas actuales también es mucho mayor, exigiendo más tiempo de estudio que antes. Aunque internet ayude mucho, seguir un orden planeado de aprendizaje es muy importante.

Al empezar a enseñar programación, el desafío siempre fue encontrar un libro que pudiese ser leído tanto por alumnos de la enseñanza media como de comienzos de la enseñanza superior. Aunque varias obras hayan suplido esa necesidad, el uso de apuntes siempre fue necesario pues el orden en que los nuevos conceptos eran presentados casi siempre era más adecuado para un diccionario que para la enseñanza de la programación en sí. Normalmente, muchos conceptos importantes para el principiante quedaban completamente de lado, y el foco principal se ponía en los asuntos más complejos. Siguiendo el enfoque utilizado por mis profesores de enseñanza media, considero que la lógica de programación es más importante que cualquier lenguaje. Quien aprende a programar una vez, se vuelve más capaz para aprender otros lenguajes de programación. Es de acuerdo a esa lógica de programación para principiantes que este libro se desarrolla, presentando recursos de Python siempre que sea posible. El propósito es iniciar al lector en el mundo de la programación y prepararlo para cursos y conceptos más avanzados. Creo que después de leer y estudiar este libro, usted estará capacitado para leer otras obras de programación y aprender nuevos lenguajes por cuenta propia.

Introducción

Este libro fue escrito teniendo en mente al principiante en programación. Aunque el lenguaje Python sea muy poderoso y rico en recursos modernos de programación, este libro no pretende solo enseñar el lenguaje en sí, sino también enseñar a programar en cualquier lenguaje. Algunos recursos del lenguaje Python no fueron utilizados porque el objetivo ha sido privilegiar los ejercicios de lógica de programación y preparar mejor al lector para otros lenguajes. Esa elección no impidió que se presentaran recursos poderosos del lenguaje, pero este libro no es una referencia del lenguaje Python.

Los capítulos fueron organizados de modo de presentar progresivamente los conceptos básicos de programación. Se recomienda leer el libro y al mismo tiempo tener a mano una computadora con el intérprete de Python abierto, para facilitar la realización de los ejemplos propuestos.

Cada capítulo trae ejercicios organizados de modo tal que sea posible verificar y examinar mejor el contenido presentado. Algunos ejercicios solo modifican los ejemplos del libro, en tanto otros requieren la aplicación de los conceptos presentados en la creación de nuevos programas. Trate de resolver los ejercicios a medida que son presentados. Aunque sea imposible no hablar de matemática en un libro de programación, los ejercicios fueron elaborados para el nivel de conocimiento de un alumno de enseñanza media, y utiliza problemas comerciales o del día a día. Esa elección no fue hecha para evitar el estudio de la matemática, sino para no mezclar la introducción de conceptos de programación con nuevos conceptos matemáticos.

Se recomienda que organice los programas generados en una carpeta (directorio) por capítulo, de preferencia agregando el número del ejemplo o ejercicio a los nombres de los archivos. Algunos ejercicios alteran otros ejercicios, aun en capítulos diferentes. Una buena organización de esos archivos va a facilitar su trabajo de estudio.

Se preparó un apéndice para ayudar a entender los mensajes de error que pueden ser generados por el intérprete de Python.

El uso de Python también libera a alumnos y profesores para utilizar el sistema operativo de su elección, ya sea Windows, Linux o Mac OS X. Todos los ejemplos del libro solo requieren la distribución estándar del lenguaje, que es ofrecida gratuitamente.

Aunque se hayan realizado todos los esfuerzos posibles para evitar errores y omisiones, no hay garantías que el libro esté exento de los mismos. Si Ud. encuentra fallas en el contenido, envíe un e-mail a errores@librodepython.com. En caso de dudas, aunque yo no pueda garantizar respuesta a todos los e-mails, envíe su mensaje a dudas@librodepython.com. Consejos, críticas y sugerencias pueden ser enviados a profesores@librodepython.com. El código-fuente y eventuales correcciones de este libro pueden ser encontrados en el sitio <http://www.librodepython.com>.

Un resumen del contenido de cada capítulo es presentado a continuación:

Capítulo 1 – Motivación: busca presentar el desafío de aprender y estimular el estudio de la programación de computadoras, presentando problemas y aplicaciones del día a día.

Capítulo 2 – Preparación del ambiente: instalación del intérprete de Python, introducción al editor de textos, presentación del IDLE, ambiente de ejecución, cómo digitar programas y hacer las primeras pruebas con operaciones aritméticas en el intérprete.

Capítulo 3 – Variables y entrada de datos: tipos de variables, propiedades de cada tipo, operaciones y operadores. Presenta el concepto de programa en el tiempo y una técnica simple de rastreo. Entrada de datos a través del teclado, conversión de tipos de datos y errores comunes.

Capítulo 4 – Condiciones: estructuras condicionales, conceptos de bloques y selección de líneas a ejecutar basadas en la evaluación de expresiones lógicas.

Capítulo 5 – Repeticiones: estructura de repetición **while**, contadores, acumuladores. Presenta el concepto de repetición de la ejecución de un bloque y de repeticiones anidadas.

Capítulo 6 – Listas: operaciones con listas, ordenación por el método de burbuja, investigación, utilización de listas como pilas y colas.

Capítulo 7 – Trabajando con cadenas de caracteres: presenta operaciones avanzadas con cadenas de caracteres. Explora la clase cadena de caracteres de Python. El capítulo incluye la realización de un juego simple para fijar los conceptos de manipulación de cadenas de caracteres.

Capítulo 8 – Funciones: noción de función y transferencia de flujo, funciones recursivas, funciones lambda, parámetros, módulos. Presenta números aleatorios.

Capítulo 9 – Archivos: creación y lectura de archivos en disco. Generación de archivos HTML en Python, operaciones con archivos y directorios, parámetros por la línea de comando, caminos.

Capítulo 10 – Clases y objetos: introducción a la orientación a objetos. Explica los conceptos de clase, objetos, métodos y herencia. Prepara al alumno para continuar estudiando el tópico y comprender mejor el asunto.

Capítulo 11 – Banco de datos: introducción al lenguaje SQL y al banco de datos SQLite.

Capítulo 12 – Próximos pasos: capítulo final, que lista los próximos pasos en diversos tópicos como juegos, sistemas web, programación funcional, interfaces gráficas y bancos de datos. Busca presentar libros y proyectos open source por medio de los cuales el alumno puede continuar estudiando, dependiendo de su área de interés.

Apéndice A – Mensajes de error: explica los mensajes de error más frecuentes en Python, mostrando sus causas y cómo resolverlos.

Motivación

Entonces, ¿Ud. quiere aprender a programar?

Programar computadoras es una tarea cuyo aprendizaje cabal exige tiempo y dedicación. Muchas veces no basta solo con estudiar y hacer los ejercicios, sino que también es necesario dejar que la mente se acostumbre a la nueva forma de pensar. Para muchas personas lo más difícil es mantener el entusiasmo por programar. Desisten ante las primeras dificultades y no vuelven más a estudiar. Otras personas son más pacientes, aprenden a no irritarse con la máquina y a asumir sus errores.

Para no sufrir de los males del que no aprendió a programar, es necesario que responda algunas preguntas antes de empezar:

1. ¿Ud. quiere aprender a programar?
2. ¿Cuál es su nivel de paciencia?
3. ¿Cuánto tiempo pretende estudiar?
4. ¿Cuál es su objetivo al programar?

1.1 ¿Ud. quiere aprender a programar?

Responda esta pregunta, pero piense un poco antes de llegar a la respuesta final. La manera más difícil de aprender a programar es no querer programar. El deseo debe venir de usted y no de un profesor o de un amigo. Programar es un arte y necesita dedicación para ser dominado. Como todo lo desconocido, es muy difícil cuando no lo entendemos, pero se vuelve más simple a medida que lo aprendemos.

Si ya decidió aprender a programar, pase a la próxima parte. Si aún no se convenció, continúe leyendo. Programar puede volverse un nuevo *hobby* y hasta una profesión. Si estudia computación, necesita saber programar. Eso no significa que será un programador toda la vida, o que la programación limitará su crecimiento dentro del área de la informática. Una excusa que ya oí muchas veces es “*yo sé programar, pero no me gusta*”. Varios alumnos de computación terminan sus cursos sin saber programar; o sea, sin saber programar realmente. Programar es como andar en bicicleta, no se olvida, pero sólo se aprende haciéndolo. Al cambiar de un lenguaje de programación a otro, si usted realmente aprendió a programar tendrá poca dificultad para aprender el nuevo lenguaje. A diferencia de saber programar, la sintaxis de un lenguaje de programación se olvida muy fácilmente. No piense que saber programar es memorizar todos esos comandos, parámetros y nombres extraños. Programar es saber utilizar un lenguaje de programación para resolver problemas, o sea, saber expresar una solución por medio de un lenguaje de programación.

1.2 ¿Cuál es su nivel de paciencia?

Sea paciente.

Otro error de quien estudia programación es querer hacer cosas difíciles desde el comienzo.

¿Cuál será su primer programa? ¿Un editor de textos? ¿Una hoja de cálculos? ¿Una calculadora?

¡No! Será algo mucho más simple... Cómo sumar dos números.

Es eso mismo: ¡sumar dos números!

Con el tiempo, la complejidad y el tamaño de los programas aumentarán.

Sea paciente.

Programar exige mucha paciencia y, principalmente, atención a los detalles. Una simple coma en lugar de un punto, u olvidar unas comillas, puede arruinar su programa. Al comienzo es común perder la calma o aún desesperarse hasta aprender a leer lo que realmente escribimos en nuestros programas. En esa etapa, la paciencia nunca es demasiada. Lea nuevamente el mensaje de error o deténgase para entender lo que no está funcionando correctamente. Nunca piense que la computadora está contra usted, ni le eche la culpa al día o al destino.

Sea paciente.

1.3 ¿Cuánto tiempo pretende estudiar?

Se puede aprender a programar en pocas horas. Si usted es el tipo de persona que programa el microondas de la tía; que para abrir los frascos de remedios lee las instrucciones de la tapa o que jugó con el Lego, entonces “programar” es su segundo nombre.

Todos nosotros ya programamos algo en la vida, aunque sea ir al cine el sábado. La cuestión es: ¿cuánto tiempo le va a dedicar a aprender a programar computadoras?

Como todo en la vida, nada de exageraciones. En realidad, tanto el tiempo como la forma de estudiar varían mucho de persona a persona. Algunas rinden más estudiando en grupo. A otras les gusta asistir a clase.

Lo importante es organizar el estudio de la programación de acuerdo a su estilo preferido. No trate de aprender todo o de entender todo rápidamente. Si eso sucede, felicitaciones, hay mucho por delante. En caso contrario, relájese. Si no entiende en el segundo intento, deje y vuelva a intentarlo mañana.

Cuando encuentre un problema, tenga calma. Vea qué escribió. Verifique si entiende lo que está escrito. Un error común es querer programar sin saber escribir las instrucciones. Es como querer escribir sin saber hablar.

Inicie el estudio con sesiones de una o dos horas por día como máximo. Después ajuste ese tiempo a su ritmo.

1.4 ¿Para qué programar?

Si Ud. no necesita programar para su trabajo o estudio, veamos algunas otras razones:

1.4.1 Escribir páginas web

Hoy, todos están expuestos a la web, la Internet y a sus miles de programas. La web sólo funciona porque permite la publicación de páginas y más páginas de textos e imágenes usando tan solo un editor de textos. La página más compleja que visitó es un conjunto de líneas de texto reunidas para instruir a un programa, el navegador (*browser*), sobre cómo presentar su contenido.

1.4.2 Poner en hora su reloj

¿Ud. conoce algunas personas que nunca aprendieron a poner en hora sus relojes? Yo recuerdo varias...

Seguir instrucciones es muy importante para tareas tan simples como esas. La secuencia de pasos para ajustar las horas, minutos y hasta la fecha de su reloj puede ser encarada como un programa. Normalmente se aprieta el botón de ajuste hasta que un número empieza a parpadear. Después puede usar un botón para cambiar la hora, o ir directamente al ajuste de los minutos. Eso se repite hasta que ha ajustado todos los valores tales como segundos, día, mes y, a veces, el año.

1.4.3 Aprender a usar mapas

¿Ya se perdió en una ciudad extraña? ¿Ya hizo una lista de pasos para llegar a algún lugar? Entonces Ud. ya programó. Sólo por buscar un mapa Ud. ya merecería un premio. Al trazar un camino desde donde está hasta donde desea llegar, relaciona una lista de calles o referencias de ese camino. Normalmente es algo así como “pasar tres calles a la izquierda”, “doblar a la derecha”, “doblar a la izquierda”... O algo como “seguir derecho hasta encontrar una señal de tránsito o un río”. O sea, programar. Seguir su programa es la mejor forma de saber si lo que escribió es correcto o si ahora está realmente perdido.

1.4.4 Mostrarle a sus amigos que sabe programar

Ésta puede ser la razón más complicada. Vamos a verla como un subproducto del aprendizaje y no como su objetivo final. Si esa es su razón para aprender a programar, es mejor continuar leyendo y conseguir otra.

Programar es un esfuerzo para realizar algo. Es una tarea que exige dedicación y que trae mucha satisfacción personal. Sus programas pueden ser buenos, pero ahí ya serán sus programas y no solamente su persona.

1.4.5 Parecer extraño

Entender qué significan miles de líneas de programa puede hacer que usted gane fama de “raro” entre los legos. Si ese es su objetivo, sepa que hay maneras más fáciles de lograrlo, tales como dejar de bañarse, dejarse crecer las uñas, tener el pelo naranja o violeta, parecer un roquero sin haber tocado nunca en una banda, etc.

Aunque buena parte de los programadores que conozco no sea exactamente lo que yo considero 100% normal, nadie lo es.

Saber programar no significa que Ud. sea loco o muy inteligente. Saber programar tampoco significa

que Ud. no sea loco o que no sea muy inteligente. Imagine que aprender a programar es como cualquier otra cosa que ya aprendió.

De todos modos, puede llegar un día en que empiece a tener pensamientos extraños, aún sin ser programador, pero no se preocupe porque cuando llegue ese día usted seguramente se dará cuenta...

1.4.6 Entender mejor cómo funciona su computadora

Programando puede empezar a entender porqué aquella operación falló o porqué el programa simplemente se cerró de improviso.

Programar también puede ayudarlo a utilizar mejor su planilla o editor de textos. El tipo de razonamiento que se aprende programando le servirá no sólo para hacer programas, sino también para usarlos.

1.4.7 Cocinar

Una vez necesité preparar una comida, pero las instrucciones estaban escritas en alemán. No sé nada de alemán. Tomé el primer diccionario que encontré y empecé a traducir las palabras principales. Con las palabras traducidas, traté de entender lo que debería hacer.

Aquella noche la cena fue solo una sopa instantánea. Una receta puede ser vista como un programa. Y como todo programa, sólo es posible seguirla si usted entiende lo que está escrito.

La simple secuencia de instrucciones no ayuda a una persona que no está en condiciones de entender sus efectos.

Para algunas personas programar es más fácil que aprender alemán (o cualquier otro idioma extranjero). Y como cualquier otra lengua, no se aprende sólo con un diccionario.

Los idiomas humanos son ricos en contextos, y cada palabra suele tener múltiples significados. La buena noticia: los lenguajes de programación están hechos para que las máquinas puedan entender lo que allí está representado. Eso significa que entender un programa es muy fácil, casi como consultar un diccionario. Otra buena noticia es que la mayoría de los lenguajes contienen conjuntos pequeños de “palabras”.

1.4.8 Salvar el mundo

Una buena razón para aprender a programar es salvar el mundo. ¡Eso mismo!

Todos los días, miles de kilos de alimento son desperdiciados o no llegan adonde deberían llegar por falta de organización. Programando usted puede ayudar a crear sistemas y aun programas que ayuden a otros a organizarse.

Otra buena acción es ayudar en un proyecto de software libre. Eso permitirá que muchas personas que no pueden pagar por programas para computadoras se beneficien de ellos sin cometer ningún delito.

1.4.9 Software libre

Por otra parte, ¿usted tiene licencia de uso para todos sus programas?

Si la respuesta es no, sepa que los programadores aprenden Linux y otros sistemas operativos mucho más rápidamente. También logran sacar mayor provecho de esos sistemas porque consiguen programarlos.

Si algo no existe, créelo. Si es malo, mejórelo.

Poner la programación en un mundo aparte puede ser la primera idea errónea que muchos tienen. La posibilidad de crear mundos dentro de las computadoras y de los programas, puede ser la segunda.

1.5 ¿Por qué Python?

El lenguaje de programación Python es muy interesante como primer lenguaje de programación debido a su simpleza y claridad. Aunque simple, es también un lenguaje poderoso que puede ser usado para administrar sistemas y desarrollar grandes proyectos. Es un lenguaje claro y objetivo, pues va directamente al punto, sin rodeos.

Python es software libre, o sea, puede ser utilizado gratuitamente gracias al trabajo de la Python Foundation¹ y de innumerables colaboradores. Puede utilizar Python prácticamente en cualquier arquitectura de computadoras o sistema operativo, como Linux², FreeBSD³, Microsoft Windows o Mac OS X⁴.

Python viene creciendo en varias áreas de la computación, como inteligencia artificial; bancos de datos; biotecnología; animación 3D; aplicaciones para móviles (celulares), juegos y aun como plataforma web. Eso explica porqué Python es famoso por tener “*batteries included*”, o sea, baterías incluidas, expresión que hace referencia a un producto completo que puede ser usado inmediatamente (¿Quién no recibió alguna vez un regalo de Navidad que vino sin pilas?). Hoy es difícil encontrar una biblioteca que no tenga *bindings* (enlaces) en Python. Ese hecho vuelve el aprendizaje del lenguaje mucho más interesante ya que aprender a programar en Python permite continuar utilizando los conocimientos adquiridos también para resolver problemas reales.

Una gran ventaja de Python es la legibilidad de los programas escritos en ese lenguaje. Otros lenguajes de programación utilizan innumerables marcaciones, como punto (.) o punto y coma (;), en el final de cada línea, además de los marcadores de comienzo y fin de bloque como llaves ({ }) o palabras especiales (**begin/end**). Esos marcadores vuelven a los programas un tanto más difíciles de leer y por suerte no son usados en Python. Veremos más sobre bloques y marcaciones en los capítulos siguientes.

Otro buen motivo para aprender Python es poder obtener resultados en poco tiempo. Como Python es un lenguaje completo, contando con bibliotecas para acceder a bancos de datos, procesar archivos XML, construir interfaces gráficas y aun juegos; podemos utilizar muchas funciones ya existentes escribiendo pocas líneas de código. Eso aumenta la productividad del programador, pues al utilizar bibliotecas usamos programas desarrollados y probados por otras personas, lo cual reduce el número de errores y permite concentrarse realmente en el problema que se quiere resolver.

Veamos un pequeño programa escrito en Python en la lista 1.1.

► Lista 1.1 – Programa Hola Mundo

```
print("¡Hola!")
```

La lista del programa 1.1 tiene solo una línea de código. La palabra **print** es una función utilizada para enviar datos a la pantalla de la computadora. Al escribir **print("Hola")**, le ordenamos a la computadora que exhiba el texto “¡Hola!” en la pantalla. Vea lo que se mostraría en la pantalla al ejecutar este programa en la computadora:

```
¡Hola!
```

Observe que las comillas (“”) no aparecen en la pantalla. Ese es uno de los detalles de la programación: necesitamos marcar o limitar el comienzo y el fin de nuestros mensajes con un símbolo, en este caso, comillas. Como podemos exhibir prácticamente cualquier texto en la pantalla, las primeras comillas indican el comienzo del mensaje y las siguientes, el fin. Al programar no podemos olvidar las limitaciones de la computadora. Una computadora no interpreta textos como los seres humanos. La máquina no logra diferenciar qué es un programa o un mensaje. Si no utilizamos las comillas, la computadora interpretará nuestro mensaje como un comando del lenguaje Python, generando un error.

El intérprete de Python es una gran herramienta para el aprendizaje del lenguaje. El intérprete es el programa que permite digitar y probar comandos escritos en Python y verificar los resultados instantáneamente. Veremos cómo utilizar el intérprete en la sección 2.2.

El lenguaje Python fue elegido para este libro porque simplifica el trabajo de aprendizaje y provee gran poder de programación. Como es un software libre, disponible prácticamente para cualquier tipo de computadora, su utilización no implica la adquisición de licencias de uso, muchas veces a un costo prohibitivo.

¡Bienvenido al mundo de la programación!

¹ <http://www.python.org>

² <http://www.kernel.org> o <http://www.ubuntu.com> para obtener el paquete completo

³ <http://www.freebsd.org>

⁴ <http://www.freebsd.org>

Preparando el ambiente

Antes de comenzar, necesitamos instalar el intérprete del lenguaje Python. El intérprete es un programa que acepta comandos escritos en Python y los ejecuta, línea por línea. Es él quien traducirá nuestros programas en un formato que pueda ser ejecutado por la computadora. Sin el intérprete de Python, nuestros programas no pueden ser ejecutados, siendo considerados sólo como texto. El intérprete también es responsable de verificar si escribimos correctamente nuestros programas, mostrando mensajes de error, en caso que encuentre algún problema.

El intérprete de Python no viene instalado con el Microsoft Windows: deberá instalarlo haciendo la descarga de internet. Si Ud. utiliza Mac OS X o Linux, probablemente ya esté instalado. Vea, a continuación, cómo verificar si tiene la versión correcta.

2.1 Instalación de Python

En este libro usamos Python versión 3.4. El lenguaje sufrió diversas modificaciones entre las versiones 2 y 3. La versión 3.4 fue elegida por permitir la correcta utilización de los nuevos recursos, además del hecho que es interesante aprender la versión más nueva. La versión 2.7 es también muy interesante, pues permite mezclar la sintaxis (forma de escribir) de Python 2 con la de Python 3. Para evitar complicaciones innecesarias, presentaremos solo las nuevas formas de escribir, y dejaremos que Python 3.4 verifique si hicimos todo correctamente. Python 3.4 aún no estaba disponible para todas las distribuciones Linux en el momento en que publicamos esta edición. Puede utilizar también Python 3.3 con todos los ejemplos del libro.

2.1.1 Windows

Python está disponible en todas las versiones de Windows. Como es un software libre, podemos bajar el intérprete de Python gratuitamente en el sitio <http://www.python.org>. El sitio debe parecerse al de la figura 2.1. Vea que, en el centro de la página, tenemos la opción **Downloads (descargas)**. Pase el mouse sobre **Downloads** y espere que se abra el menú.

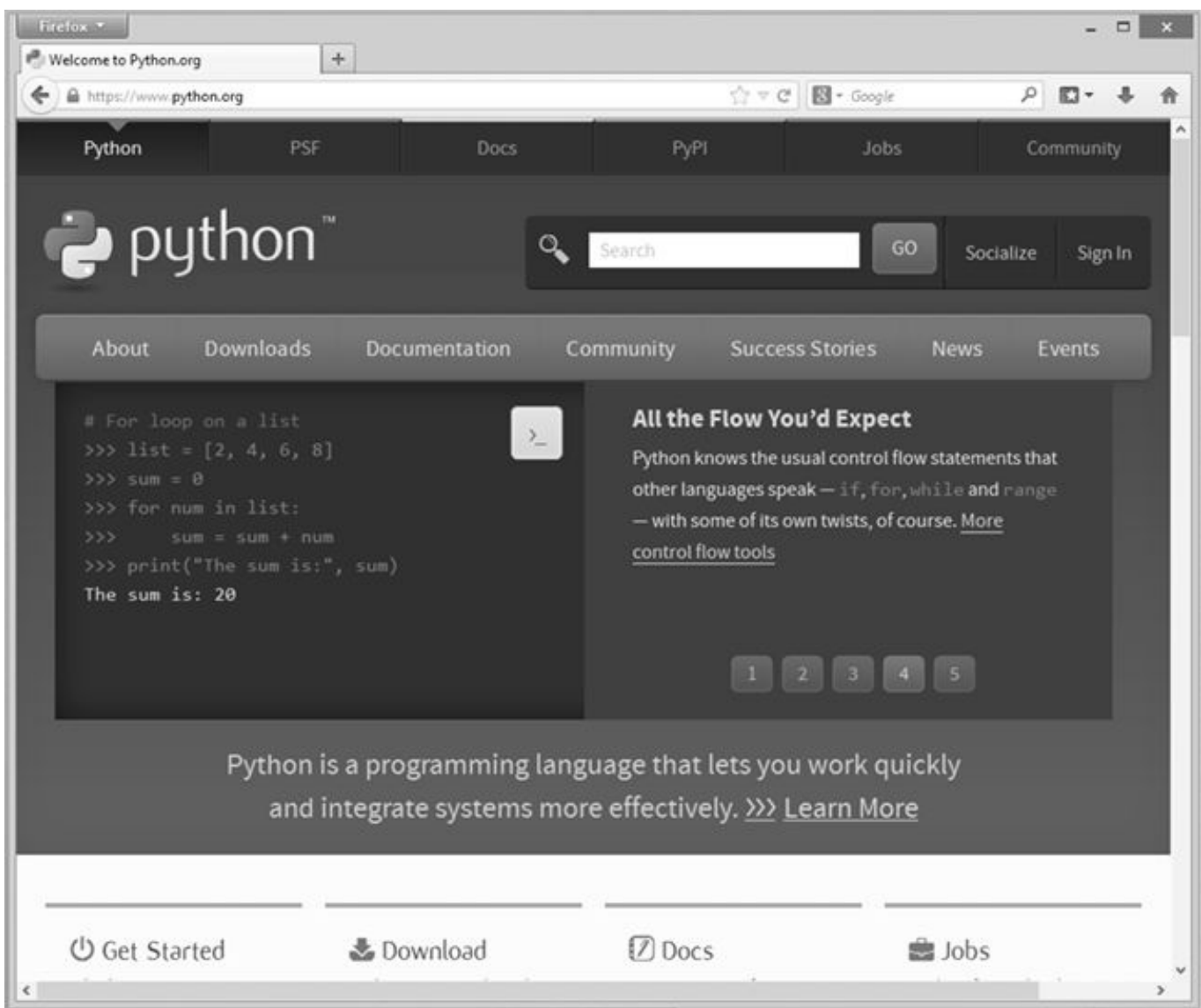


Figura 2.1 – Sitio de la Python Foundation.

La figura 2.2 muestra abierto el menú de la página de “download”. Busque el texto Python 3.4.0. Si una versión más nueva de la serie 3 está disponible, debe elegirla. Cuando este libro fue escrito, la versión más nueva era la 3.4.0. Haga clic en el botón para comenzar la descarga del archivo.

Si no está utilizando el Firefox, el menú de download puede no aparecer. En ese caso, haga clic en Windows y siga las instrucciones en las páginas que se abrirán. Si su versión de Windows es Windows Vista, 7, 8 o 8.1 32 bits, elija Python 3.4.0 Windows x86 MSI. En caso que utilice Windows Vista, 7, 8 o 8.1 64 bits, elija Python 3.4.0 Windows x86-64 MSI Installer. Si no encuentra esos archivos, trate de digitar la dirección completa de la página de descarga: <https://www.python.org/downloads/release/python-340/>.

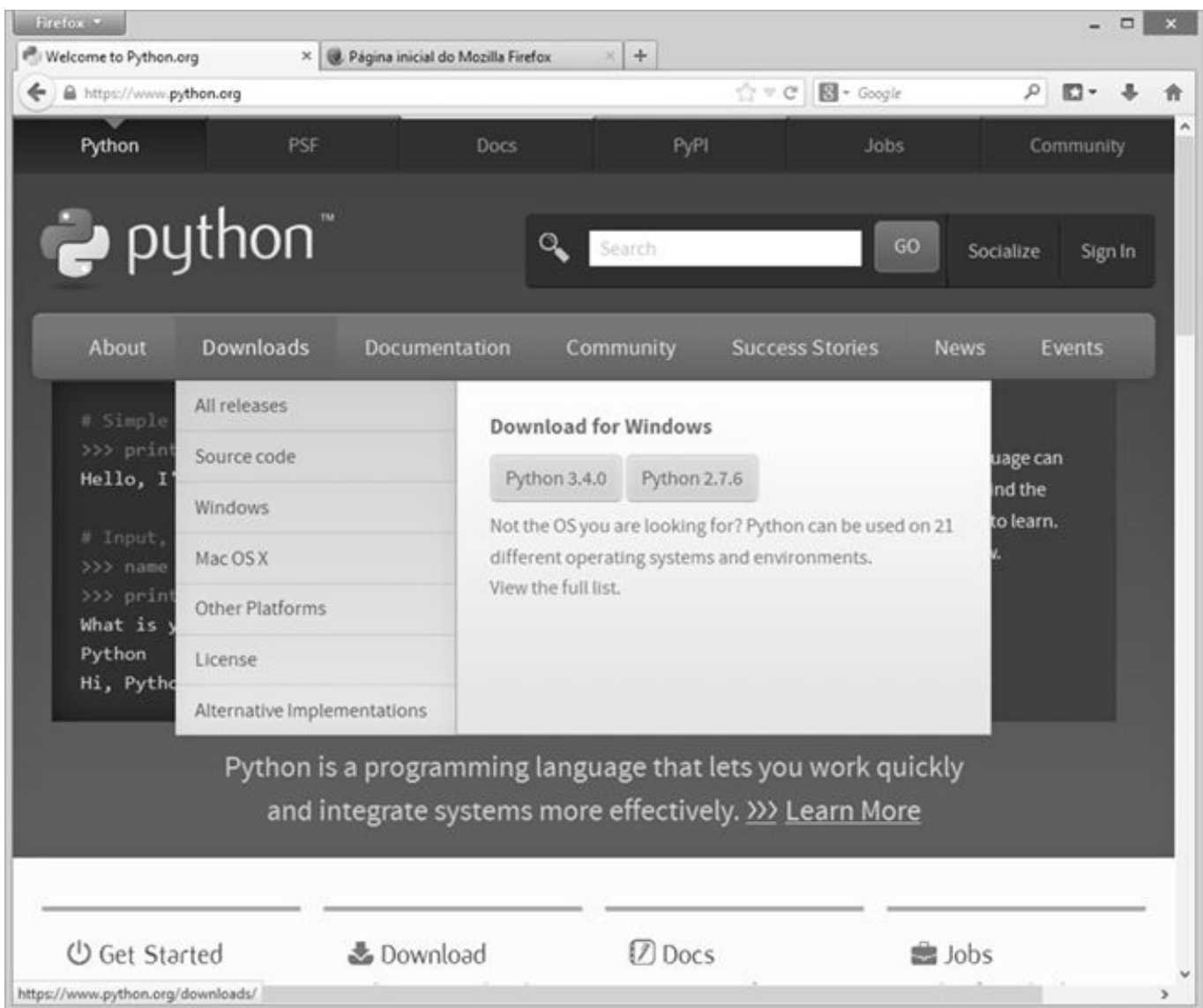


Figura 2.2 – Página de descarga.

En los ejemplos a continuación, haremos la descarga de la versión 3.4.0 para Windows 8.1. La versión de Python puede cambiar, y el nombre del archivo, también. Si eligió la versión de 32 o 64 bits, el nombre del archivo será un poco diferente, pero no se preocupe: si elige la versión equivocada, simplemente no funcionará. Puede entonces intentar con otra versión.

La figura 2.3 muestra la ventana de descarga del Firefox. Esa ventana también puede variar de acuerdo con la versión de su navegador de internet. Haga clic en el botón **download** para comenzar la transferencia del archivo. Aquí, el Firefox fue utilizado como ejemplo; pero puede utilizar cualquier otro navegador de internet, como Internet Explorer, Google Chrome, Opera o Safari.

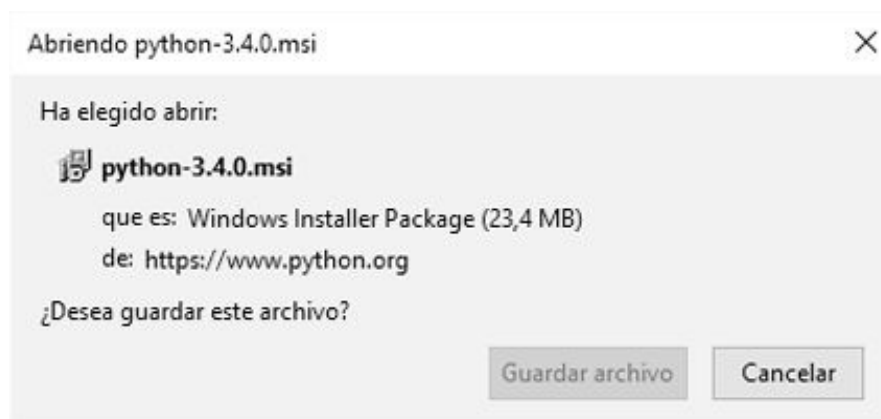


Figura 2.3 – Ventana de confirmación de descarga del Firefox.

El intérprete de Python ocupa aproximadamente 24 MB en el disco (antes de la instalación), y la transferencia del archivo debe demorar algunos minutos, dependiendo de la velocidad de conexión a internet. El Firefox exhibe la descarga como en la figura 2.4.

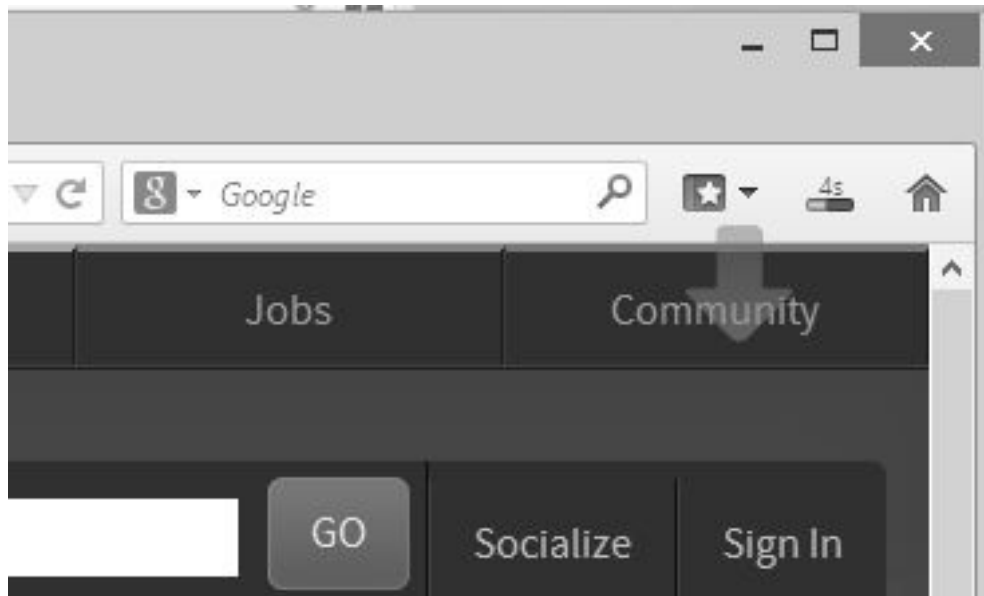


Figura 2.4 – Ventana de descarga con archivo python-3.4.0.msi siendo bajado.

Vea que la barra de transferencia desaparece cuando la transferencia está concluida, como en la figura 2.5.

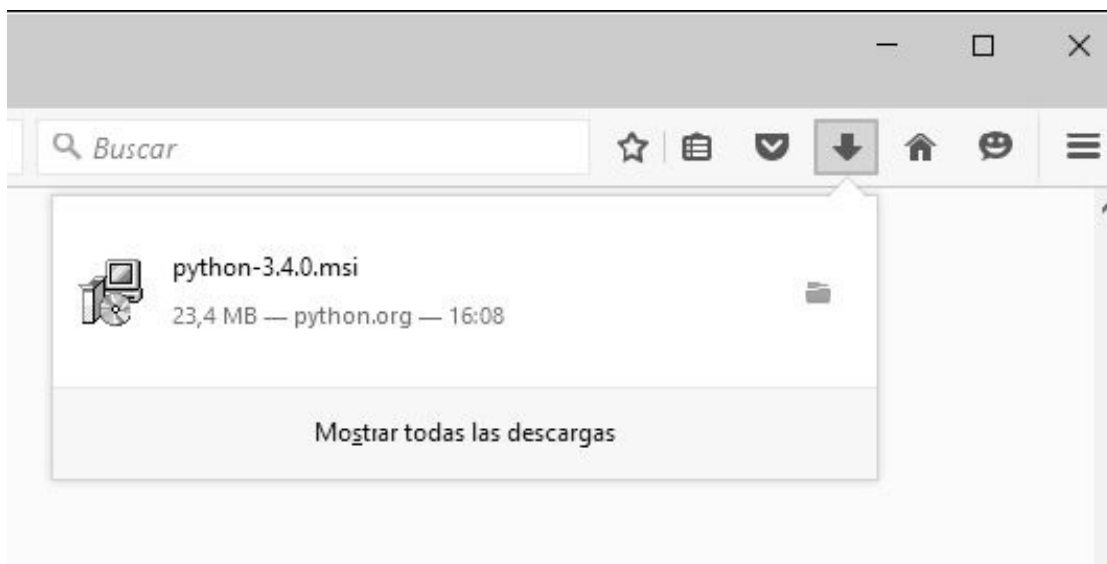


Figura 2.5 – Ventana de descarga con la transferencia del archivo finalizada.

Haga clic en el nombre del archivo que acaba de bajar. Dependiendo de la versión de su Windows y de su navegador de internet, podrá recibir un pedido de confirmación de Firefox o de Windows. En caso que Firefox pregunte si desea ejecutar el programa, elija ejecutar. Windows también puede pedir una confirmación de instalación, ilustrada en la figura 2.6. Elija “Sí” para aprobar la instalación. Esa ventana puede aparecer un poco después del comienzo de la instalación. Verifique también si un ícono parecido a un escudo coloreado empieza a parpadear; en ese caso, haga clic en el ícono para que la ventana de permiso aparezca.

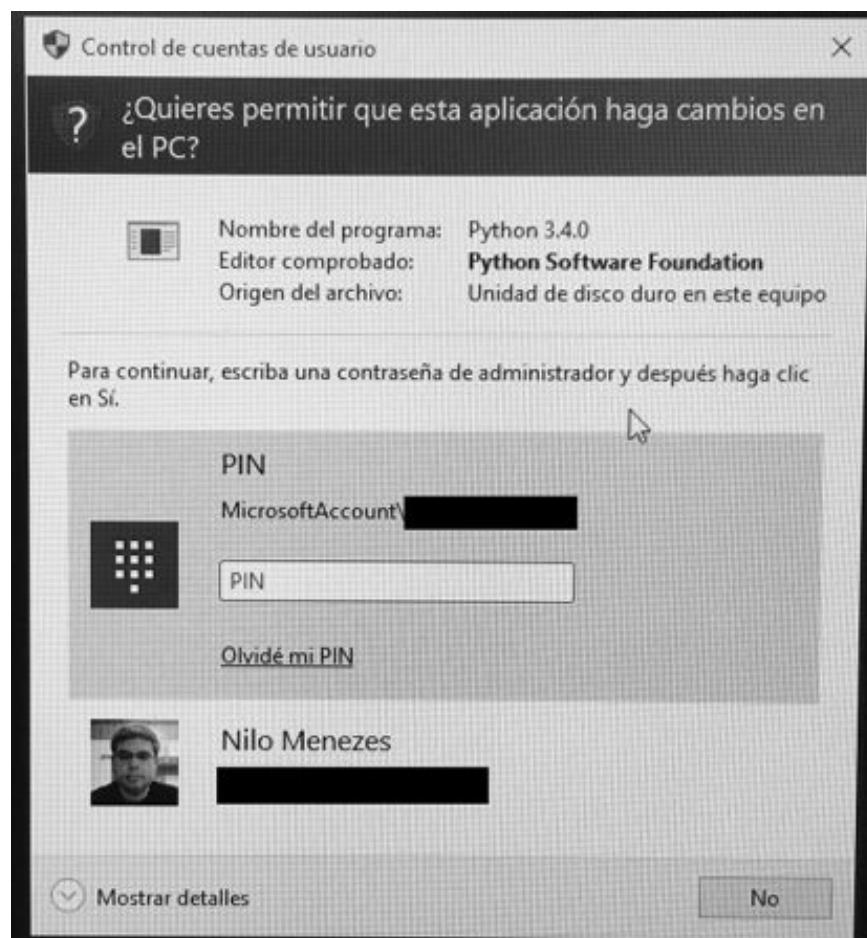


Figura 2.6 – Ventana de confirmación de la ejecución del intérprete de Python.

La figura 2.7 muestra la pantalla inicial del intérprete. Hasta ahora, no instalamos nada. Es ese programa el que va a instalar Python 3 en su computadora. Haga clic en el botón **Next** para continuar.



Figura 2.7 – Ventana del intérprete de Python.

Usted debe estar en una ventana parecida a la de la figura 2.8. Esa ventana permite elegir dónde

instalar los archivos del intérprete de Python. Puede cambiar la carpeta o simplemente aceptar el lugar propuesto (estándar). Haga clic en el botón **Next (siguiente)** para continuar.



Figura 2.8 – Ventana del intérprete pidiendo confirmación de la carpeta (directorio) de instalación de Python.

La ventana de la figura 2.9 permite seleccionar qué partes del paquete de Python queremos instalar. Esa ventana es útil para usuarios más avanzados del lenguaje, entonces simplemente vamos a aceptar el estándar, haciendo clic en el botón **Next** para continuar.

Finalmente empezamos a instalar los archivos que necesitamos. Una barra de progreso como la de la figura 2.10 será exhibida. Aguarde la finalización del proceso.



Figura 2.9 – Ventana de selección de paquetes del intérprete de Python.



Figura 2.10 – Ventana del intérprete exhibiendo el progreso de la instalación.

La instalación terminó. Haga clic en el botón **Finish** (terminar), que se ve en la figura 2.11, para salir del intérprete.



Figura 2.11 – Ventana de finalización del intérprete.

2.1.2 Linux

Gran parte de las distribuciones Linux incluye una versión del intérprete de Python; sin embargo, las versiones más utilizadas son la 2.6 o la 2.7. Necesitamos el intérprete en la versión 3.3 o superior. Para verificar la versión de su intérprete, digite **python -V** en la línea de comando.

En Ubuntu (versión 14.04), digite:

```
sudo apt-get install python3.4
sudo apt-get install idle-python3.4
```

Necesita permisos de administrador (root) para efectuar la instalación. Si la versión 3.4 aún no está disponible para instalación vía apt-get, puede utilizar la versión 3.3 sin problemas. Consulte el sitio del libro para más detalles.

2.1.3 Mac OS X

Los Mac OS X Leopard, Snow Leopard, Lion, Mountain Lion, Maverick, Yosemite Y El Capitan vienen con una versión del intérprete de Python de Apple. Sin embargo, esa versión no es la 3.4. Para resolver el problema, instale el MacPorts (<http://www.macports.org/>), haciendo el download del archivo dmg, y, después, instale Python 3.4 con:

```
sudo port install python34
```

Para ejecutar el intérprete recién instalado, digite **python3.4** en la línea de comando. También puede utilizar otros administradores de paquetes disponibles para Mac OS X, como Fink (<http://www.finkproject.org/>) y Homebrew (<http://brew.sh/>). Una versión instalable, en formato dmg, también está disponible en el sitio <http://www.python.org>.

2.2 Usando el intérprete

Con Python instalado, vamos a empezar a trabajar.

El IDLE es una interfaz gráfica para el intérprete de Python, que también permite la edición y ejecución de nuestros programas. En Windows Vista y en Windows 7, debe tener una carpeta en el menú **Comenzar > Programas > Python 3.4**. Elija **IDLE**. En Windows 8 y 8.1, busque por Python 3.4 en la lista de aplicaciones y, entonces, por IDLE (Python GUI).

En Linux, abra la terminal y digite:

```
idle-python3.4 &
```

En Mac OS X, abra la terminal y digite:

```
IDLE3.4 &
```

La ventana inicial del IDLE, en Windows 8.1, se muestra en la figura 2.12. Si utiliza Mac OS X, Linux o una versión diferente de Windows, esa ventana no será exactamente igual a la de la figura, pero sí muy parecida.

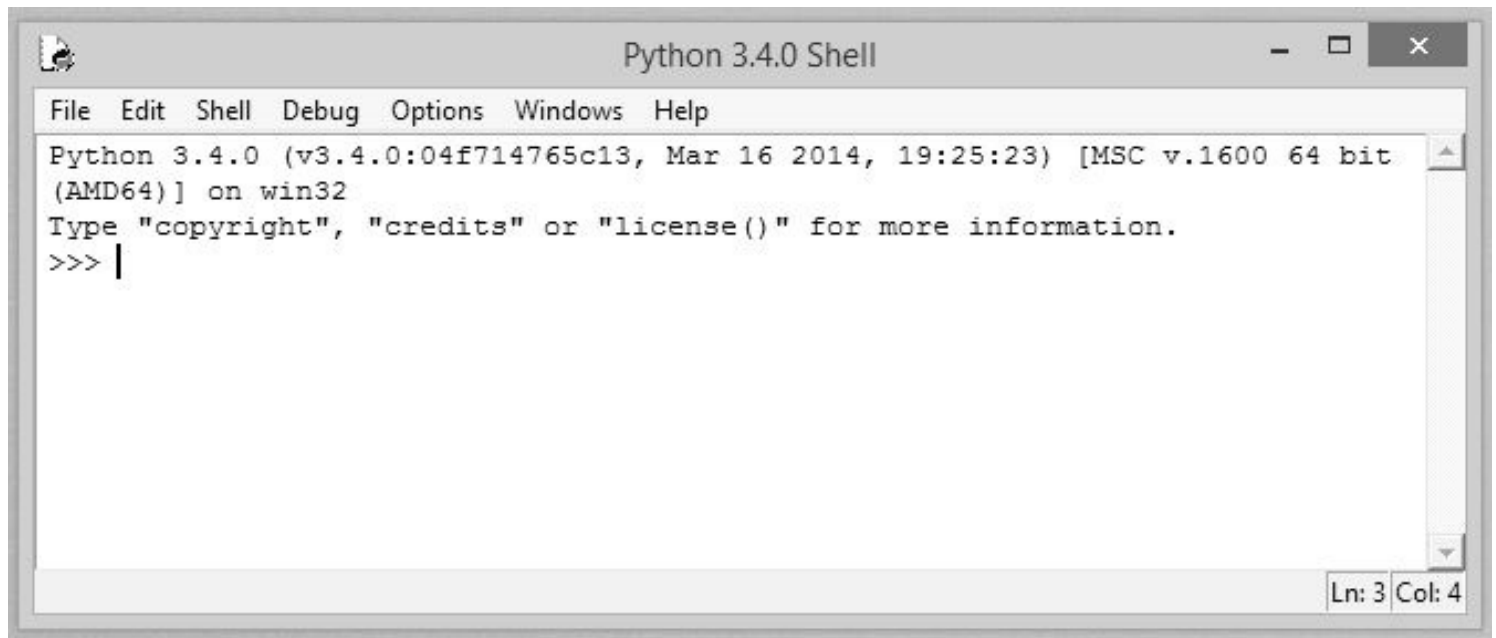


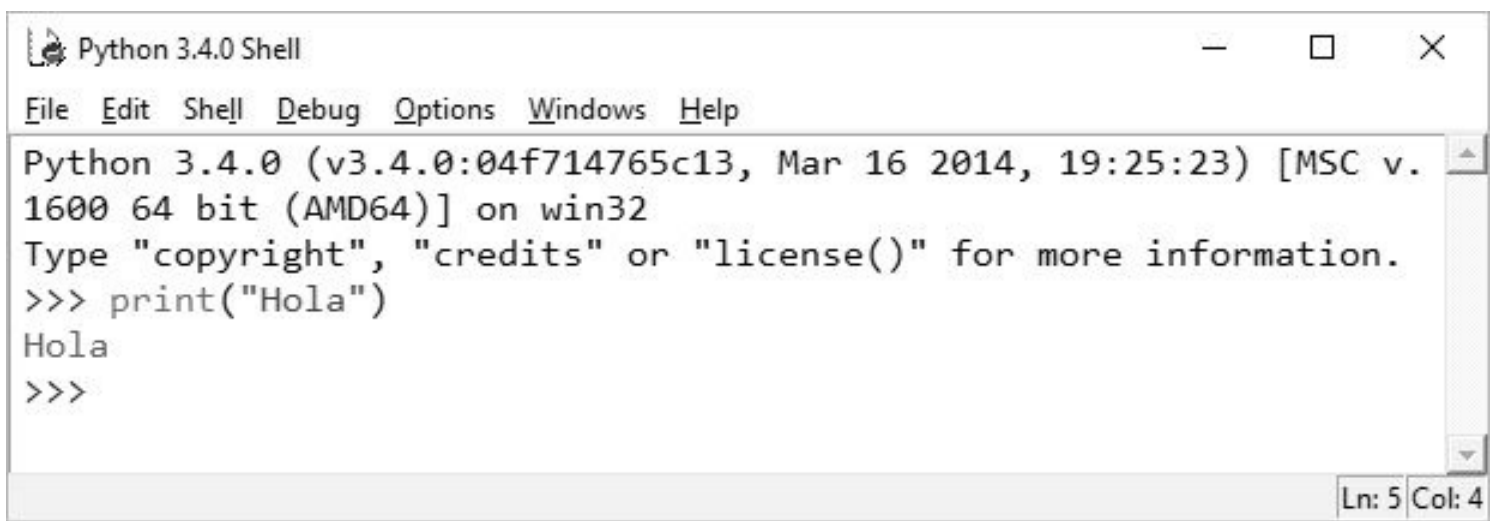
Figura 2.12 – Ventana inicial de IDLE.

Observe que el cursor esté posicionado dentro de la ventana Python Shell, y que la línea de comandos esté encabezada por la secuencia **>>>**.

Digite:

```
print("Hola")
```

Presione la tecla **ENTER**. Ud. debe obtener una pantalla parecida a la de la figura 2.13.



```
Python 3.4.0 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:25:23) [MSC v.
1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("Hola")
Hola
>>>
```

Figura 2.13 – IDLE mostrando el resultado de `print("Hola")`.

Una de las grandes ventajas de Python es contar con el intérprete de comandos. Puede abrir una ventana como esa siempre que tenga alguna duda en Python. El intérprete permite que Ud. verifique el resultado de un comando instantáneamente. Veremos más adelante que ese tipo de verificación es muy importante y facilita el aprendizaje del lenguaje.

2.3 Editando archivos

No sólo de experimentos vive el programador de Python. Un programa es un archivo de texto, escrito en un formato especial (lenguaje).

En el caso del lenguaje Python, podemos usar cualquier editor de textos disponible: PSPad o Sublime, en Windows; TextMate, en Mac OS X; Vim o Emacs, en Linux.

Lo que no debe utilizar es un editor de textos como el Microsoft Word o el OpenOffice. Esos programas graban sus archivos en formatos especiales que no pueden ser utilizados para escribir programas, salvo si elige la opción “guardar solo texto” o simplemente grabar en el formato txt. Además de eso, un editor de textos común no fue hecho para escribir programas.

Puede también utilizar el editor de textos incluido en la instalación del intérprete de Python. Con el intérprete abierto, haga clic en el menú **File (archivo)** y después seleccione la opción **New Window**, como muestra la figura 2.14.

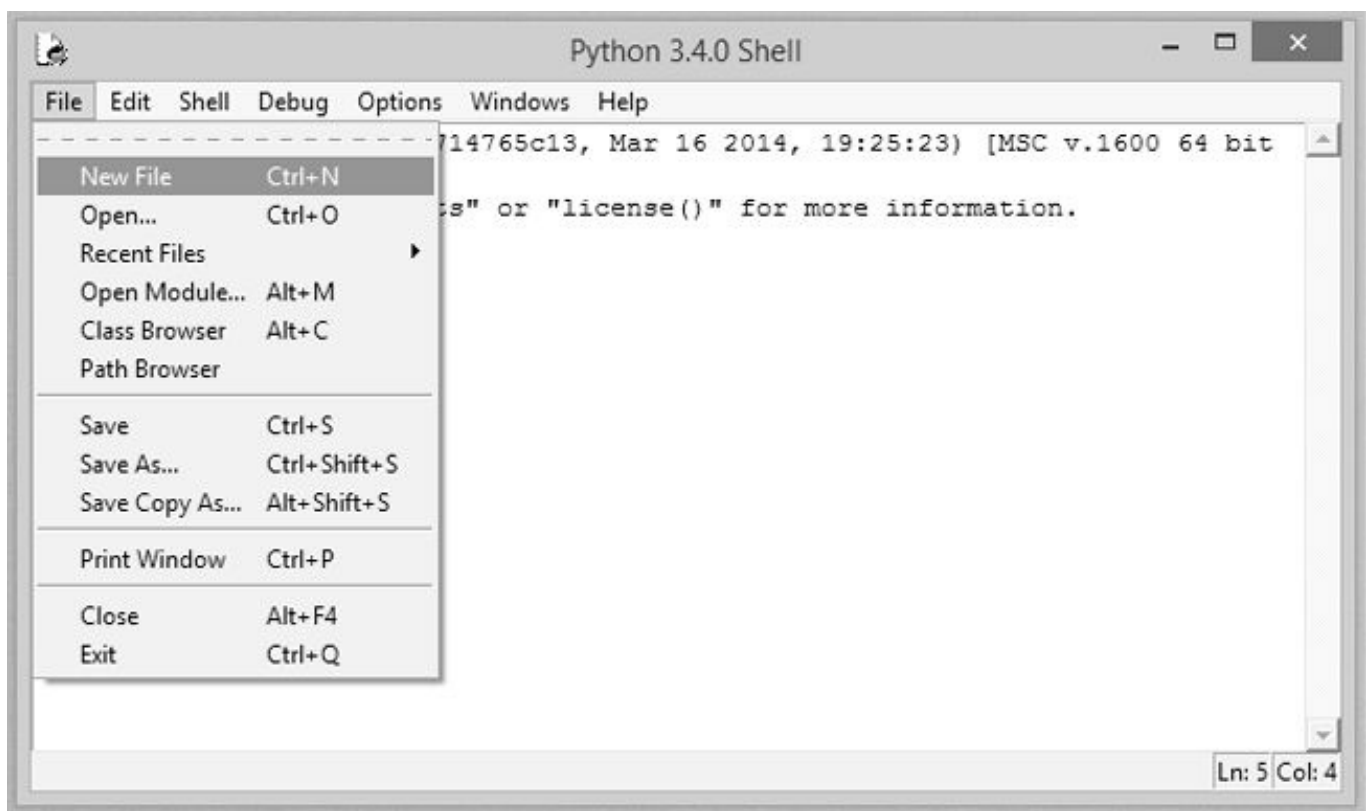


Figura 2.14 – IDLE con el menú File abierto y la opción New Window seleccionada.

Como utilizamos la versión 3.4 del intérprete, es muy importante que use un editor de textos correcto. O sea, un editor que soporte la edición de textos en UTF-8. Si Ud. no sabe qué es UTF-8, o si su editor de textos no soporta ese formato de codificación, utilice el IDLE como editor de textos. Además de soportar la edición en el formato UTF-8, que permitirá la utilización de acentos en nuestros programas, está preparado para colorear en Python, volviendo más fácil la lectura.

Una nueva ventana, como la de la figura 2.15, deberá aparecer. Es en esa ventana que escribiremos nuestros programas. Observe que, aunque sea parecida a la ventana principal del IDLE, la ventana tiene opciones de menú diferentes de la otra. Para aclarar esa separación, llamaremos a la primera de ventana del intérprete; y a la segunda, de ventana del editor de textos. Si Ud. aún está en duda, la ventana del editor de textos es la que presenta la opción **Run** en el menú.

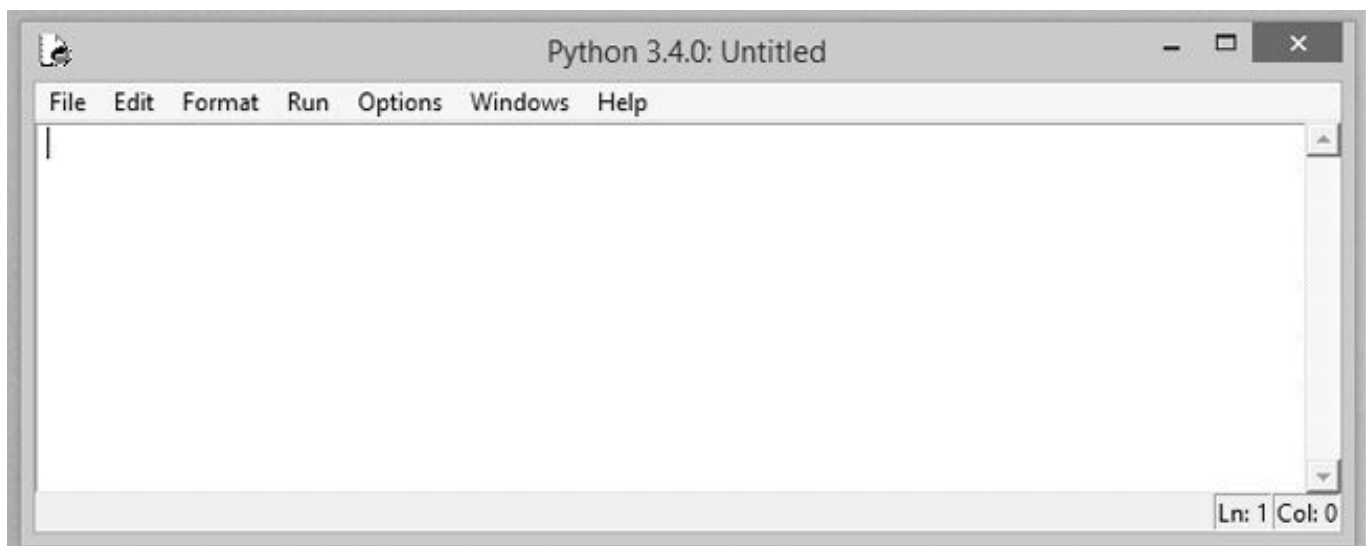


Figura 2.15 – Ventana del editor de textos del IDLE.

Pruebe un poco, escribiendo:

```
print("Hola")
```

Su ventana del editor de textos deberá ser semejante a la mostrada en la figura 2.16.

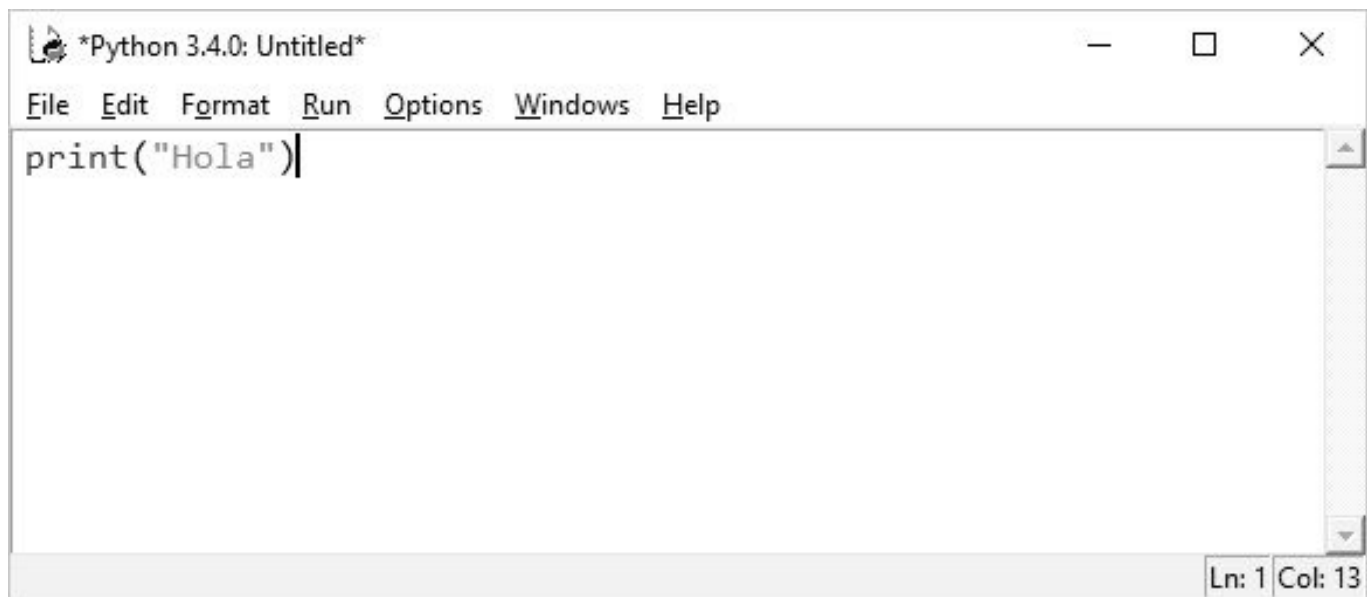


Figura 2.16 – Editor de textos con el programa ya digitado.

Ahora que escribimos nuestro pequeño programa, vamos a guardarlo. Elija la opción **Save** del menú **File**, como muestra la figura 2.17.

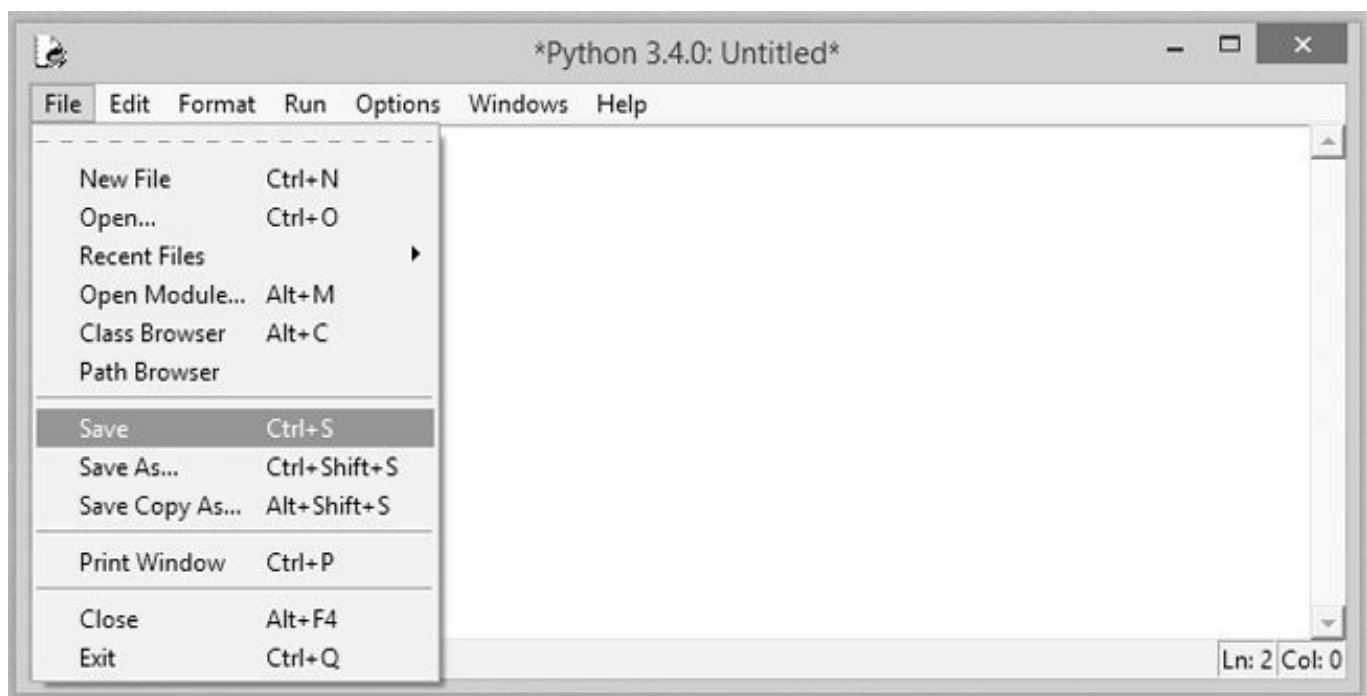


Figura 2.17 – Ventana del editor de textos con el menú File abierto y la opción Save seleccionada.

Una ventana estándar de grabación de archivos será exhibida. La ventana cambia con la versión del sistema operativo, pero si Ud. utiliza Windows 8.1, se parecerá a la de la figura 2.18. Escriba `test.py` en el nombre del archivo y haga clic en el botón para guardar. Atención: la extensión `.py` no es adicionada automáticamente por el IDLE. Recuerde siempre grabar sus programas escritos en Python con la extensión `.py`.

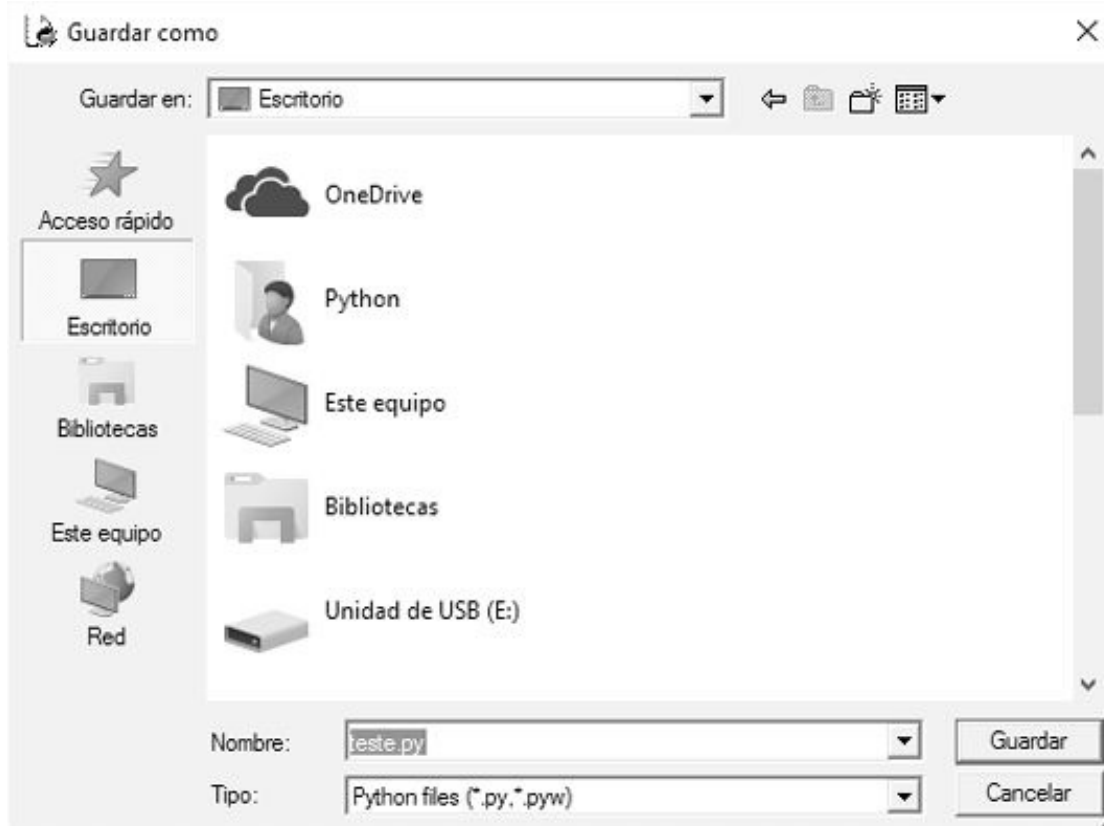


Figura 2.18 – Ventana de grabación de archivos de Windows 8.1.

Ahora haga clic en el menú **Run** para ejecutar su programa. Puede también presionar la tecla F5 con el mismo efecto. Esa operación puede ser visualizada en la figura 2.19.

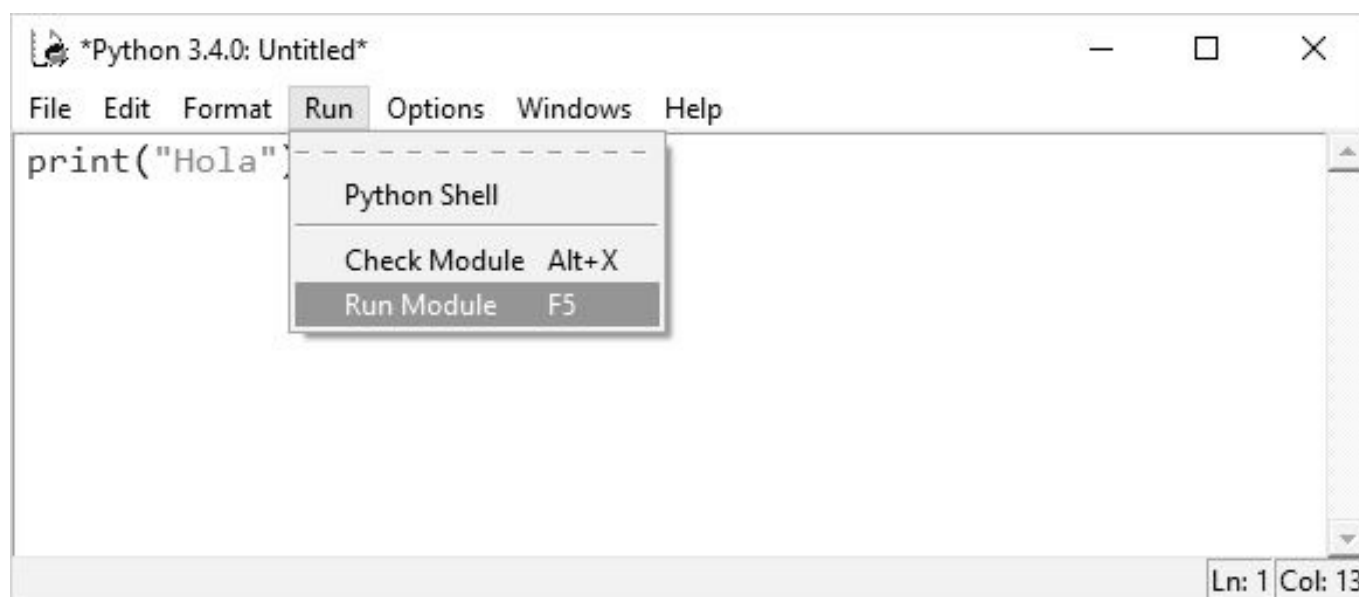


Figura 2.19 – Ventana del intérprete mostrando la opción Run Module seleccionada.

Su programa será ejecutado en la otra ventana, la del intérprete. Vea que una línea con la palabra RESTART apareció, como en la figura 2.20. Observe que obtuvimos el mismo resultado de nuestro primer test.



```
Python 3.4.0 Shell
File Edit Shell Debug Options Windows Help
Python 3.4.0 (v3.4.0:04f714765c13, Mar 16 2014, 19:25:23) [MSC v.1600 64 bit
(AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("Hola")
Hola
>>> ===== RESTART =====
>>>
Hola
>>>
```

Figura 2.20 – Ventana del intérprete mostrando la ejecución del programa.

Pruebe modificar el texto entre comillas y ejecutar nuevamente el programa, presionando la tecla **F5** en la ventana del Editor o seleccionando **Run** en el menú.

Ahora estamos con todo instalado para continuar aprendiendo a programar. Durante todo el libro se presentarán programas de ejemplo. Debe digitarlos en la ventana del editor de textos, grabarlos y ejecutarlos. Puede volver a leer las secciones 2.2 y 2.3 siempre que lo necesite. Con el tiempo, esas tareas serán memorizadas y realizadas sin esfuerzo.

Pruebe grabar sus programas en un directorio específico en su computadora. Eso ayudará a encontrarlos más tarde. Un consejo es crear un directorio para cada capítulo del libro, pero puede organizar los ejemplos como quiera. Puede también hacer copias de los ejemplos y alterarlos sin problema. Aunque pueda bajar todos los ejemplos ya digitados, es altamente recomendado que lea el libro y digite cada programa. Esa práctica ayuda a memorizar las construcciones del lenguaje y facilita la lectura de los programas.

2.4 Cuidados al digitar sus programas

Al digitar un programa en el editor de textos, verifique si lo copió exactamente como es presentado. En Python, debe tener cuidado con los siguientes ítems:

1. Las letras mayúsculas y minúsculas son diferentes. Así, **print** y **Print** son completamente diferentes, causando un error en caso que digite la P mayúscula. Al comienzo, es común que el hábito nos lleve a escribir la letra inicial de cada línea en mayúscula, causando errores en nuestros programas. En caso de error, lea atentamente lo que digitó y compárelo con la lista presentada en el libro.
2. Las comillas son muy importantes y no deben ser olvidadas. Cada vez que abra comillas, no se olvide de cerrarlas. Si se olvida, su programa no funcionará. Observe que el IDLE cambia el color del texto entre comillas, facilitando esa verificación.
3. Los paréntesis no son opcionales en Python. No remueva los paréntesis de los programas y présteles la misma atención que a las comillas, para abrirlas y cerrarlas. Todo paréntesis abierto debe ser cerrado.
4. Los espacios son muy importantes. El lenguaje Python se basa en la cantidad de espacio en

blanco antes del comienzo de cada línea para realizar diversas operaciones, explicadas posteriormente en el libro. No se olvide de digitar el texto de los programas con la misma alineación presentada en el libro. Observe también que el IDLE ayuda en esos casos, avisando sobre problemas de alineación. Nunca junte dos líneas en una sola hasta sentirse seguro sobre cómo escribir correctamente en Python.

2.5 Los primeros programas

Vamos a analizar nuestro primer programa. La figura 2.21 muestra la separación entre el nombre de la función, los paréntesis, el mensaje y las comillas. Es muy importante saber el nombre de cada una de esas partes para el correcto entendimiento de los conceptos y programas presentados.

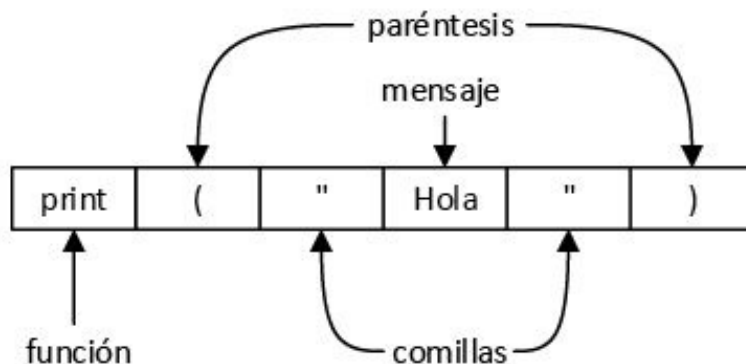


Figura 2.21 – Nombres de los elementos del primer programa.

La función **print** informa que vamos a exhibir algo en la pantalla. Se puede decir que la función muestra un mensaje en la pantalla de la computadora. Siempre que queramos mostrarle algo al usuario de la computadora, como un mensaje, una pregunta o el resultado de una operación de cálculo, utilizaremos la función **print**.

Para separar los programas del texto explicativo del libro, todo tramo de programa; funciones; variables y demás ítems en Python serán presentados con una fuente de texto diferente (monoespaciada), como la utilizada para explicar la función **print**.

Volviendo a la figura 2.21, tenemos las comillas como elementos importantes. Las mismas son utilizadas para separar textos destinados al usuario de la computadora del resto del programa. Utilizamos comillas para indicar el comienzo y el fin del texto de nuestro mensaje. El mensaje es nuestro texto en sí y se mostrará exactamente como ha sido digitado. Los paréntesis son utilizados para separar los parámetros de una función, en este caso, los de **print**. Un parámetro es un valor pasado a una función: en el caso de la función **print**, el mensaje a imprimir.

El intérprete de Python también puede ser utilizado como calculadora. Pruebe digitar `2+3` en el intérprete, como muestra la lista 2.1.

► Lista 2.1 – Usando el intérprete como calculadora

```
>>> 2+3
```

```
5
```

No se olvide de presionar a tecla **Enter** para que el intérprete sepa que terminó la digitación. El resultado debe ser presentado en la línea siguiente. Podemos también substraer valores, como en la

lista 2.2.

► Lista 2.2 – Sustracción

```
>>> 5-3  
2
```

Podemos combinar adición y sustracción en la misma línea, como muestra la lista 2.3.

► Lista 2.3 – Adición y sustracción

```
>>> 10-4+2  
8
```

La multiplicación es representada por un asterisco (*); y la división, por la barra (/), como muestra la lista 2.4.

► Lista 2.4 – Multiplicación y división

```
>>> 2*10  
20  
>>> 20/4  
5.0
```

Para elevar un número a un exponente, utilizaremos dos asteriscos (**) para representar la operación de exponenciación. Observe que no hay ningún espacio entre los dos asteriscos. Así, para calcular 2^3 , escribiremos de forma semejante al contenido de la lista 2.5.

► Lista 2.5 – Exponenciación

```
>>> 2**3  
8
```

Podemos también obtener el resto de la división de dos números usando el símbolo %. Así, para calcular el resto de la división entre 10 y 3, digitaríamos tal como se muestra en la lista 2.6.

► Lista 2.6 – Resto de la división entera

```
>>> 10 % 3  
1  
>>> 16 % 7  
2  
>>> 63 % 8  
7
```

Los paréntesis son utilizados en Python del mismo modo que en expresiones matemáticas, o sea, para alterar el orden de ejecución de una operación. El orden de precedencia de las operaciones, respeta las siguientes prioridades:

1. Exponenciación o potenciación (**).

2. Multiplicación (*) y división (/ y %).

3. Adición (+) y sustracción (-).

La expresión $1500 + (1500 * 5 / 100)$ es equivalente a:

$$1500 + \left(\frac{1500 \times 5}{100} \right)$$

No se olvide que tanto en Python como en matemática, las operaciones de la misma prioridad son realizadas de izquierda a derecha. Utilice paréntesis siempre que necesite alterar el orden de ejecución de las operaciones y también para aumentar la claridad de la fórmula.

Ejercicio 2.1 Convierta las siguientes expresiones matemáticas para que puedan ser calculadas usando el intérprete de Python.

$10 + 20 \times 30$

$4^2 \div 30$

$(9^4 + 2) \times 6 - 1$

Ejercicio 2.2 Digite la siguiente expresión en el intérprete:

`10 % 3 * 10 ** 2 + 1 - 10 * 4 / 2`

Trate de resolver el mismo cálculo, usando sólo lápiz y papel. Observe como la prioridad de las operaciones es importante.

2.6 Conceptos de variables y atribución

Además de operaciones simples de cálculo, el intérprete también puede ser usado para realizar operaciones más complejas y aún ejecutar programas completos. Antes de continuar, es importante observar el concepto de variables y cómo podemos usarlas en un programa. En matemática, aprendemos el concepto de variable para representar incógnitas en ecuaciones del tipo $x + 1 = 2$, donde debemos determinar el valor de x , resolviendo la ecuación. En programación, las variables son utilizadas para almacenar valores y para darle nombre a un área de memoria de la computadora donde almacenamos datos. Las variables serán mejor estudiadas en el capítulo 3. Por ahora, podemos imaginar la memoria de la computadora como un gran estante, donde cada compartimiento tiene un nombre. Para almacenar algo en esos compartimento, usaremos el símbolo de igualdad (=) entre el nombre del compartimiento y el valor que queremos almacenar. Llamaremos a esa operación “de atribución”, donde un valor es atribuido a una variable. Cuando leamos nuestro programa, las operaciones de atribución serán llamadas de “recibe”, o sea, una variable recibe un valor.

A fin de simplificar las explicaciones sobre cómo funciona un programa, utilizaremos bolas negras ❶ con números para relacionar una determinada línea a un texto explicativo. Esos símbolos no hacen parte del programa y no deben ser digitados ni en el intérprete ni en el editor de textos.

Como casi todo en la vida, se aprende a programar programando. Vamos a escribir otro programa. Observe la lista 2.7:

► Lista 2.7 – El primer programa con variables

```
a = 2 ❶  
b = 3 ❷  
print(a + b) ❸
```

Veamos qué significa cada línea. En ❶, tenemos `a = 2`. Léase “a recibe 2”. Esa línea dice que una variable llamada `a` recibirá el valor 2. Las variables en programación tienen el mismo significado que en matemática. Puede entender una variable como una forma de guardar valores en la memoria de la computadora. Toda variable necesita tener un nombre para que su valor pueda ser utilizado posteriormente. Ese concepto quedará más claro un poco más adelante.

En ❷, tenemos `b = 3`. Lea “b recibe 3”. Esa línea realiza un trabajo muy parecido al de la línea anterior, pero la variable se llama `b`, y el valor es el número 3. Para entender qué hace esa línea, imagine que creamos un espacio en la memoria de la computadora para guardar otro valor, en este caso, 3. Para poder usar ese valor más tarde, llamamos “b” a ese espacio.

La línea ❸ solicita que el resultado de la suma del contenido de la variable `a` con el contenido de la variable `b` sea exhibido en la pantalla. La función `print` realiza la impresión, pero antes, el resultado de `a + b` es calculado. Vea que en esa línea estamos ordenándole al programa que calcule `a + b` y que exhiba el resultado en la pantalla. Como en matemática, pasamos parámetros o valores para una función usando paréntesis. Esos paréntesis son requeridos por el intérprete de Python. Se debe recordar que la notación es $f(x)$; donde f es el nombre de la función, y x un parámetro. En el ejemplo anterior, `print` es el nombre de la función; y el resultado de `a + b`, el valor pasado como parámetro. En el transcurso de este libro, veremos diversas funciones disponibles en Python para realizar operaciones con la computadora, tales como leer valores del teclado o grabar datos en un archivo.

Puede probar el programa de la lista 2.7 en la ventana del intérprete de Python, como muestra la sección 2.2. El resultado de ese programa se presenta en la lista 2.8.

► Lista 2.8 – Ejemplo mostrado en el intérprete

```
>>> a = 2  
>>> b = 3  
>>> print( a + b )  
5
```

Las dos primeras líneas no envían nada a la pantalla; por eso solo se muestra el resultado de la tercera línea.

Puede estar preguntándose porqué creamos dos variables, `a` y `b`, para sumar dos números. Podríamos haber obtenido el mismo resultado de diversas formas, como en la lista 2.9.

► Lista 2.9 – Otra forma de resolver el problema

```
print(2 + 3)
```

O también como muestra la lista 2.10.

► Lista 2.10 – Otra forma de resolver el problema

```
print(5)
```

Entonces, ¿por qué elegimos resolver el problema usando variables? Primero, para poder hablar de variables, pero también para ejemplificar una gran diferencia entre resolver un problema en el papel y por medio de una computadora. Cuando estamos resolviendo un problema de matemática en el papel, como sumar dos números, realizamos diversos cálculos mentalmente y escribimos parte de ese proceso en el papel, cuando es necesario. Después de escrito en el papel, cambiar los valores no es tan simple. Al programar una computadora, estamos transfiriendo ese cálculo a la computadora. Como programar es describir los pasos para la solución del problema, es aconsejable escribir programas lo más claramente posible, de modo que podamos alterarlos en el caso que lo precisemos y, más importante aún, para poder entenderlos más tarde.

En la lista 2.9, el problema fue representado como la suma de 2 y 3. Si necesitamos cambiar las partes de esa suma, tendremos que escribir otro programa. Eso también es válido para el primer programa, pero observe que al utilizar variables, estamos dándole nombre a los valores de entrada de nuestro problema, aumentando, así, la facilidad de entender lo que hace el programa.

La solución presentada en la lista 2.10 no describe el problema en sí. Estamos sólo ordenándole a la computadora que imprima el número 5 en la pantalla. No hicimos ningún registro de lo que estábamos haciendo, o que nuestro problema era sumar dos números. Eso quedará más claro en el ejemplo a continuación. Vea la lista 2.11.

► Lista 2.11 – Cálculo de aumento de salario

```
salario = 1500 ❶  
aumento = 5 ❷  
print(salario + (salario * aumento / 100)) ❸
```

En ❶ tenemos una variable que es llamada `salario`, recibiendo el valor 1500. En ❷, otra variable, `aumento`, recibe el valor 5. Finalmente, en ❸ describimos la fórmula que calculará el valor del nuevo salario después de recibir un aumento. Tendríamos, entonces, un resultado como el de la lista 2.12.

► Lista 2.12 – Resultado del aumento de salario en el intérprete

```
>>> salario = 1500  
>>> aumento = 5  
>>> print(salario + (salario * aumento / 100))  
1575.0
```

El programa de la lista 2.11 puede ser escrito de forma más directa, utilizando otra fórmula sin variables. Vea la alternativa de la lista 2.13.

► Lista 2.13 – Alternativa para el cálculo de aumento de salario

```
print(1500 + (1500 * 5 / 100))
```

El objetivo de ese ejemplo es presentar la diferencia entre describir el problema de forma genérica, separando los valores de entrada del cálculo. El resultado es idéntico: la diferencia está en la claridad de la representación de nuestro problema. Si cambiamos el valor del salario, en la primera línea de la lista 2.11, obtendremos el resultado correcto en la salida del programa, sin necesidad de

preocuparnos nuevamente por la fórmula del cálculo. Observe también que si hacemos lo mismo en el programa de la lista 2.13, tendremos que cambiar el valor del salario en dos posiciones diferentes de la fórmula, aumentando nuestras posibilidades de olvidarnos de una de ellas y, consecuentemente, de recibir un resultado incorrecto.

Al utilizar variables, podemos referenciar el mismo valor varias veces, sin olvidarnos que podemos utilizar nombres más significativos que una simple “x” o “y”, para aumentar la claridad del programa. Por ejemplo, en la lista 2.11, registramos la fórmula para el cálculo del aumento especificando el nombre de cada variable, facilitando así la lectura y la comprensión.

Si Ud. ya utilizó una hoja de cálculos, como Microsoft Excel u OpenOffice Calc, el concepto de variable puede ser entendido como las células de una hoja de cálculos. Puede escribir las fórmulas de su planilla sin utilizar otras células, pero tendría que reescribirlas cada vez que los valores cambiasen. Así como las células de una hoja de cálculos, las variables de un programa pueden ser utilizadas varias veces y en lugares diferentes.

Ejercicio 2.3 Haga un programa que exhiba su nombre en la pantalla.

Ejercicio 2.4 Escriba un programa que exhiba el resultado de $2a \times 3b$, donde a vale 3 y b vale 5.

Ejercicio 2.5 Modifique el primer programa, lista 2.7, de modo que calcule la suma de tres variables.

Ejercicio 2.6 Modifique el programa de la lista 2.11, de modo que calcule un aumento del 15% para un salario de € 750.

Variables y entrada de datos

El capítulo anterior presentó el concepto de variables, pero hay más por descubrir. Ya sabemos que las variables tienen nombres que permiten acceder a los valores de esas variables en otras partes del programa. En este capítulo vamos a ampliar nuestro conocimiento sobre variables, estudiando nuevas operaciones y nuevos tipos de datos.

3.1 Nombres de las variables

En Python, los nombres de las variables deben comenzar obligatoriamente con una letra, pero pueden contener números y el símbolo guión bajo (`_`). Veamos ejemplos de nombres válidos e inválidos en Python en la tabla 3.1.

Tabla 3.1 – Ejemplo de nombres válidos e inválidos para variables

Nombre	Válido	Comentarios
a1	Sí	Aunque contenga un número, el nombre a1 empieza con una letra.
velocidad	Sí	Nombre formado por letras.
velocidad90	Sí	Nombre formado por letras y números, pero iniciado por una letra.
salario_medio	Sí	El símbolo guión bajo (<code>_</code>) está permitido y facilita la lectura de nombres grandes.
salario medio	No	Los nombres de las variables no pueden contener espacios en blanco.
b	Sí	El guión bajo (<code></code>) es aceptado en nombres de variables, aún en el comienzo.
1a	No	Los nombres de las variables no pueden empezar con números.

La versión 3 del lenguaje Python permite el uso de acentos en los nombres de las variables pues, por estándar, los programas son interpretados utilizando un conjunto de caracteres llamado UTF-8 (<http://es.wikipedia.org/wiki/Utf-8>), capaz de representar prácticamente todas las letras de los alfabetos conocidos.

Las variables tienen otras propiedades además de nombre y contenido. Una de ellas es conocida como “tipo” y define la naturaleza de los datos que almacena la variable. Python tiene varios tipos de datos, pero los más comunes son números enteros, números de punto flotante y cadenas de caracteres. Además de poder almacenar números y letras, las variables en Python también almacenan valores como verdadero o falso. Decimos que esas variables son de tipo lógico. Veremos más sobre variables de tipo lógico en la sección 3.3.

TRIVIA

La mayor parte de los lenguajes de programación fue desarrollada en los Estados Unidos, considerando

solo nombres escritos en la lengua inglesa como nombres válidos. Por eso, acentos y letras consideradas especiales no son aceptados como nombres válidos en la mayor parte de los lenguajes de programación. Esa restricción es un problema no solo para hablantes del español, sino de muchas otras lenguas que utilizan acentos y aun otros alfabetos. El estándar americano está basado en el conjunto de caracteres ASCII ^(*), desarrollado en la década de 1960. Con la globalización de la economía y el surgimiento de internet, las aplicaciones más modernas son escritas para trabajar con un conjunto de caracteres universal, llamado Unicode ^(**).

(*) <http://es.wikipedia.org/wiki/Ascii>
(**) <http://es.wikipedia.org/wiki/Unicode>

3.2 Variables numéricas

Decimos que una variable es numérica cuando almacena números enteros o de punto flotante.

Los números enteros son aquellos sin parte decimal, como 1, 0, -5, 550, -47, 30000.

Números de punto flotante o decimales son aquellos con parte decimal, como 1.0, 5.478, 10.478, 30000.4. Observe que 1.0, aun teniendo cero en la parte decimal, es un número de punto flotante.

En Python, y en la mayoría de los lenguajes de programación, utilizamos el punto, y no la coma, como separador entre la parte entera y la fraccionaria de un número. Esa es otra herencia de la lengua inglesa. Observe también que no utilizamos nada como separador de millar. Ejemplo: 1.000.000 (un millón) es escrito 1000000.

Ejercicio 3.1 Complete la tabla a continuación, marcando entero o punto flotante de acuerdo al número presentado.

Número	Tipo numérico
5	<input type="checkbox"/> entero <input type="checkbox"/> punto flotante
5.0	<input type="checkbox"/> entero <input type="checkbox"/> punto flotante
4.3	<input type="checkbox"/> entero <input type="checkbox"/> punto flotante
-2	<input type="checkbox"/> entero <input type="checkbox"/> punto flotante
100	<input type="checkbox"/> entero <input type="checkbox"/> punto flotante
1.333	<input type="checkbox"/> entero <input type="checkbox"/> punto flotante

3.2.1 Representación de valores numéricos

Internamente, todos los números son representados utilizando el sistema binario, o sea, de base 2. Ese sistema permite sólo los dígitos 0 y 1. Para representar números mayores, combinamos varios dígitos, exactamente como hacemos con el sistema decimal, o de base 10, que utilizamos normalmente.

Veamos primero cómo funciona eso en un sistema con base 10:

$$\begin{aligned} 531 &= 5 \times 10^2 + 3 \times 10^1 + 1 \times 10^0 \\ &= 5 \times 100 + 3 \times 10 + 1 \times 1 \\ &= 500 + 30 + 1 \\ &= 531 \end{aligned}$$

Multiplicamos cada dígito por la base elevada a un exponente igual al número de casas a la derecha del dígito en cuestión. Como en 531 el 5 tiene 2 dígitos a la derecha, multiplicamos 5×10^2 . Para el 3, tenemos solo otro dígito a la derecha, luego multiplicamos 3×10^1 . Finalmente, para el 1, sin dígitos a la derecha, tenemos 1×10^0 . Sumando esos componentes, tenemos el número 531. Hacemos eso tan rápido que es natural o automático pensar de ese modo.

El cambio al sistema binario sigue el mismo proceso, pero la base ahora es 2, y no 10. Así, 1010 en binario representa:

$$\begin{aligned} 1010 &= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ &= 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 \\ &= 8 + 2 \\ &= 10 \end{aligned}$$

La utilización del sistema binario es transparente en Python, o sea, si usted no solicita explícitamente que ese sistema sea usado, todo será presentado en la base 10 utilizada en el día a día. La importancia de la noción de diferencia de base es mucha, pues ella explica los límites de la representación. El límite de representación está dado por el valor mínimo y el valor máximo que pueden ser representados en una variable numérica. Ese límite está condicionado por la cantidad de dígitos que fueron reservados para almacenar el número en cuestión. Veamos cómo funciona en la base 10.

Si usted tiene solo 5 dígitos para representar un número, el número mayor es 99999, y el menor sería (-99999). Lo mismo sucede en el sistema binario, siendo que allí reservamos un dígito para registrar las señales de positivo y negativo.

Para números enteros, Python utiliza un sistema de precisión ilimitada que permite la representación de números muy grandes. Es como si Ud. siempre pudiese escribir nuevos dígitos a medida que sea necesario. Puede calcular en Python valores como $2^{1000000}$ (`2 ** 1000000`) sin problemas de representación, aún cuando el resultado es un número de 301030 dígitos.

En cuanto al punto flotante, tenemos límites y problemas de representación. Un número decimal es representado en punto flotante utilizando una mantisa y un exponente (*señal* \times *mantisa* \times $base^{\text{exponente}}$). Tanto la mantisa como el exponente tienen un número máximo de dígitos que limita los números que pueden ser representados. No necesita preocuparse por eso en este momento, pues esos valores son muy grandes y no tendrá problemas en la mayoría de sus programas. Puede obtener más información entrando a http://es.wikipedia.org/wiki/Coma_flotante.

La versión 3.1.2 de Python, en Mac OS X, tiene como límites $2.2250738585072014 \times 10^{-308}$ y $1.7976931348623157 \times 10^{308}$, suficientemente grandes y pequeños para casi cualquier tipo de aplicación. Es posible encontrar problemas de representación en función de cómo los números decimales son convertidos en números de punto flotante. Esos problemas son bien conocidos y afectan a todos los lenguajes de programación, no es un problema específico de Python.

Veamos un ejemplo: el número 0.1 no tiene nada de especial en el sistema decimal, pero es un decimal periódico en el sistema binario. No necesita preocuparse por esos detalles ahora, pero puede investigarlos más tarde cuando lo necesite (normalmente los cursos de computación tienen una disciplina llamada cálculo numérico, para abordar esos tipos de problemas). Digite en el intérprete `3 * 0.1`

Debe haber obtenido como resultado 0.30000000000000004 y no 0.3, como era de esperar. No se asuste: no es un problema de su computadora, sino de representación. Si durante sus estudios necesita cálculos más precisos, o si los resultados con punto flotante no satisfacen los requisitos de precisión esperados, verifique los módulos **decimal** y **fractions**. La documentación de Python trae una página específica sobre ese tipo de problema: <https://docs.python.org/3/tutorial/float.html>.

3.3 Variables del tipo lógico

Muchas veces, queremos almacenar un contenido simple: verdadero o falso en una variable. En ese caso, utilizaremos un tipo de variable llamado tipo lógico o booleano. En Python, escribiremos **True** para verdadero y **False** para falso (Lista 3.1). Observe que T y el F se escriben con letras mayúsculas.

► **Lista 3.1 – Ejemplo de variables del tipo lógico**

```
resultado = True
aprobado = False
```

3.3.1 Operadores de comparación

Para realizar comparaciones lógicas, utilizaremos operadores de comparación. La lista de operadores de comparación soportados en Python es presentada en la tabla 3.2.

Tabla 3.2 – Operadores de comparación

Operador	Operación	Símbolo matemático
<code>==</code>	igualdad	<code>=</code>
<code>></code>	mayor que	<code>></code>
<code><</code>	menor que	<code><</code>
<code>!=</code>	diferente	<code>≠</code>
<code>>=</code>	mayor o igual	<code>≥</code>
<code><=</code>	menor o igual	<code>≤</code>

El resultado de una comparación es un valor del tipo lógico, o sea, **True** (verdadero) o **False** (falso). Utilizaremos el verbo “evaluar” para indicar la resolución de una expresión.

► **Lista 3.2 – Ejemplo de uso de operadores de comparación**

```
>>> a = 1      # a recibe 1
>>> b = 5      # b recibe 5
>>> c = 2      # c recibe 2
>>> d = 1      # d recibe 1
>>> a == b     # ¿a es igual a b?
False
>>> b > a      # ¿b es mayor que a?
True
```

```
>>> a < b      # ¿a es menor que b?
True
>>> a == d     # ¿a es igual a d?
True
>>> b >= a     # ¿b es mayor o igual a a?
True
>>> c <= b     # ¿c es menor o igual a b?
True
>>> d != a     # ¿d es diferente de a?
False
>>> d != b     # ¿d es diferente de b?
True
```

Esos operadores son utilizados como en matemática. Especial atención debe ser dada a los operadores `>=` y `<=`. El resultado de esos operadores es realmente mayor o igual y menor o igual, o sea, `5 >= 5` es verdadero, así como `5 <= 5`.

Observe que, en la lista 3.2, utilizamos el símbolo de numeral (`#`) para escribir comentarios en la línea de comando. Todo texto a la derecha del numeral es ignorado por el intérprete de Python, o sea, puede escribir lo que quiera. Lea nuevamente la lista 3.2 y vea cómo es más fácil entender cada línea cuando la comentamos. Los comentarios no son obligatorios, pero son muy importantes. Puede y debe utilizarlos en sus programas para facilitar su comprensión y disponer de una anotación para Ud. mismo. Es común leer un programa algunos meses después de escrito y tener dificultades para recordar qué queríamos hacer realmente.

No necesita comentar todas las líneas de sus programas o escribir lo obvio. Un consejo es identificar los programas con su nombre, la fecha en que empezó a ser escrito y también la lista o capítulo del libro donde lo encontró.

Variables del tipo lógico también pueden ser utilizadas para almacenar el resultado de expresiones y comparaciones.

► Lista 3.3 – Ejemplo del uso de operadores de comparación con variables del tipo lógico

```
nota = 8
media = 7
aprobado = nota > media
print(aprobado)
```

Si una expresión contiene operaciones aritméticas, éstas deben ser calculadas antes que los operadores de comparación sean evaluados. Cuando evaluamos una expresión, sustituimos el nombre de las variables por su contenido y sólo entonces verificamos el resultado de la comparación.

Ejercicio 3.2 Complete la tabla de abajo, respondiendo True o False. Considere $a = 4$, $b = 10$, $c = 5.0$, $d = 1$ y $f = 5$.

--	--

Expresión	Resultado
a == c	[] True [] False
a < b	[] True [] False
d > b	[] True [] False
c != f	[] True [] False
a == b	[] True [] False
c < d	[] True [] False
b > a	[] True [] False
c >= f	[] True [] False
f >= c	[] True [] False
c <= c	[] True [] False
c <= f	[] True [] False

3.3.2 Operadores lógicos

Para agrupar operaciones con lógica booleana, utilizaremos operadores lógicos. Python soporta tres operadores básicos: **not** (no), **and** (e), **or** (o). Esos operadores pueden ser traducidos como “no” (\neg negación), “y” (\wedge conjunción) y “o” (\vee disyunción).

Tabla 3.3 – Operadores lógicos

Operador Python	Operación
not	no
and	y
or	o

Cada operador obedece a un conjunto simple de reglas, expresado por la tabla verdad de ese operador. La tabla verdad demuestra el resultado de una operación con uno o dos valores lógicos u operandos. Cuando el operador utiliza solo un operando, decimos que es un operador unario. Al utilizar dos operandos, es llamado operador binario. El operador de negación (**not**) es un operador unario; **or** (o) y **and** (e) son operadores binarios, necesitando, así, de dos operandos.

3.3.2.1 Operador not

El operador **not** (no) es el más simple, pues necesita sólo un operador. La operación de negación también es llamada de inversión, pues un valor verdadero negado se vuelve falso y viceversa. La tabla verdad del operador **not** (no) es presentada en la tabla 3.4.

Tabla 3.4 – Tabla verdad del operador not (no)

V_1	$\text{not } V_1$
V	F
F	V

► Lista 3.4 – Operador not

```
>>>not True
False
```

```
>>>not False
True
```

3.3.2.2 Operador and

El operador **and** (y) tiene su tabla verdad representada en la tabla 3.5. El operador **and** (y) resulta verdadero solo cuando sus dos operadores son verdaderos.

Tabla 3.5 – Tabla verdad del operador and (y)

V ₁	V ₂	V ₁ and V ₂
V	V	V
V	F	F
F	V	F
F	F	F

► Lista 3.5 – Operador and

```
>>>True and True
True
>>>True and False
False
>>>False and True
False
>>>False and False
False
```

3.3.2.3 Operador or

La tabla verdad del operador **or** (o) es presentada en la tabla 3.6. La regla fundamental del operador **or** (o) es que resulta falso sólo si sus dos operadores también son falsos. Si uno solo de sus operadores es verdadero, o si los dos lo son, el resultado de la operación será verdadero.

Tabla 3.6 – Tabla verdad del operador or(o)

V ₁	V ₂	V ₁ or V ₂
V	V	V
V	F	V
F	V	V
F	F	F

► Lista 3.6 – Operador or

```
>>> True or True
True
>>> True or False
True
>>> False or True
```

True
>>> False or False
False

Ejercicio 3.3 Complete la tabla a continuación utilizando a = True, b = False y c = True.

Expresión	Resultado
a and a	[] True [] False
b and b	[] True [] False
not c	[] True [] False
not b	[] True [] False
not a	[] True [] False
a and b	[] True [] False
b and c	[] True [] False
a or c	[] True [] False
b or c	[] True [] False
c or a	[] True [] False
c or b	[] True [] False
c or c	[] True [] False
b or b	[] True [] False

3.3.3 Expresiones lógicas

Los operadores lógicos pueden ser combinados en expresiones lógicas más complejas. Cuando una expresión tiene más de un operador lógico, se evalúa el operador **not** (no) primero, seguido del operador **and** (y) y, finalmente, **or** (o). Veamos a continuación el orden de evaluación de la expresión, donde la operación que está siendo evaluada aparece subrayada; y el resultado se muestra en la línea siguiente.

True or False and <u>not True</u>
True or False and <u>False</u>
<u>True or False</u>
True

Los operadores de comparación también pueden ser utilizados en expresiones con operadores lógicos.

salario > 1000 and edad > 18

En esos casos, los operadores de comparación deben ser evaluados primero. Supongamos **salario = 100** y **edad = 20**. Tendremos:

salario > 1000 and edad > 18
<u>100 > 1000</u> and <u>20 > 18</u>
<u>False and True</u>
False

La gran ventaja de escribir ese tipo de expresiones es representar condiciones que pueden ser evaluadas con valores diferentes. Por ejemplo: imagine que **salario > 1000 and edad > 18** sea una condición para un préstamo de compra de un auto nuevo. Cuando **salario = 100** y **edad =**

20, sabemos que el resultado de la expresión es falso, y podemos interpretar que, en ese caso, la persona no recibirá el préstamo. Evaluemos la misma expresión con **salario = 2000** y **edad = 30**.

```
salario > 1000 and edad > 18
2000 > 1000 and 30 > 18
True and True
True
```

Ahora el resultado es **True** (verdadero) y podríamos decir que la persona cumple con las condiciones para obtener el préstamo.

Ejercicio 3.4 Escriba una expresión para determinar si una persona debe o no pagar impuesto. Considere que pagan impuesto personas cuyo salario sea mayor que € 1.200,00.

Ejercicio 3.5 Calcule el resultado de la expresión $A > B$ and C or D , utilizando los valores de la tabla a continuación.

A	B	C	D	Resultado
1	2	True	False	
10	3	False	False	
5	1	True	True	

Ejercicio 3.6 Escriba una expresión que será utilizada para decidir si un alumno fue o no aprobado. Para ser aprobado, todas las medias del alumno deben ser mayores que 7. Considere que el alumno cursa solo tres materias, y que la nota de cada una está almacenada en las siguientes variables: `materia1`, `materia2` y `materia3`.

3.4 Variables del tipo de cadena de caracteres

Las variables del tipo cadena de caracteres almacenan cadenas de caracteres tales como nombres y textos en general. Llamamos cadena de caracteres a una secuencia de símbolos como letras, números, señales de puntuación, etc. Ejemplo: Juan y María comen pan. En ese caso, Juan es una secuencia con las letras J, u, a, n. Para simplificar el texto, utilizaremos el nombre cadena de caracteres para mencionar variables del tipo de cadenas de caracteres. Podemos imaginar una cadena de caracteres como una secuencia de bloques, donde cada letra, número o espacio en blanco ocupa una posición, como muestra la figura 3.1.

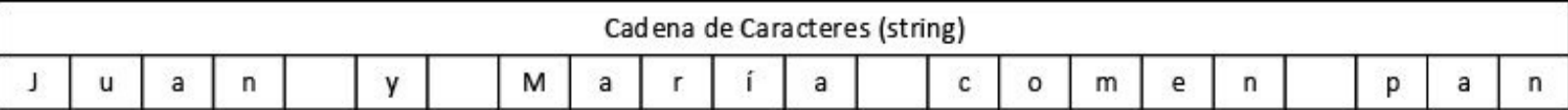


Figura 3.1 – Representación de una cadena de caracteres.

Para posibilitar la separación entre el texto del programa y el contenido de una cadena de caracteres, utilizaremos comillas (") para delimitar el comienzo y el fin de la secuencia de caracteres. Volviendo al ejemplo anterior, escribiremos **"Juan y María comen pan"**. Vea que en ese caso no hay ningún problema en utilizar espacios. En verdad, la computadora ignora prácticamente todo que escribimos entre comillas, pero veremos más tarde que no es exactamente así.

Las variables del tipo cadena de caracteres son utilizadas para almacenar secuencias de caracteres, normalmente utilizadas en textos o mensajes. El tipo cadena de caracteres es muy útil y bastante utilizado para exhibir mensajes y también para generar otros archivos.

Una cadena de caracteres en Python tiene un tamaño asociado, así como un contenido al que se puede acceder carácter por carácter. El tamaño de una cadena de caracteres puede ser obtenido utilizando la función `len`. Esa función retorna el número de caracteres en la cadena de caracteres. Decimos que una función retorna un valor cuando podemos sustituir el texto de la función por su resultado. La función `len` retorna un valor del tipo entero, representando la cantidad de caracteres contenidos en la cadena de caracteres. Si la cadena de caracteres está vacía (representada simplemente por `"`, o sea dos comillas sin nada entre ellas, ni siquiera espacios en blanco), su tamaño es igual a cero. Hagamos algunas pruebas, como muestra la lista 3.7.

► Lista 3.7 – La función `len`

```
>>> print(len("A"))
1
>>> print(len("AB"))
2
>>> print(len(""))
0
>>> print(len("Erre con erre barril"))
19
```

Como fue dicho anteriormente, otra característica de las cadenas de caracteres es poder acceder a su contenido carácter por carácter. Sabiendo que una cadena de caracteres tiene un determinado tamaño, podemos acceder a sus caracteres utilizando un número entero para representar su posición. Ese número es llamado índice, y empezamos a contar de cero. Eso quiere decir que el primer carácter de la cadena de caracteres es de posición o índice 0. Observe la figura 3.2.

Cadena de Caracteres(String)									
0	1	2	3	4	5	6	7	8	← Índice
A	B	C	D	E	F	G	H	I	← Contenido

Figura 3.2 – Índices y contenido de una variable cadena de caracteres.

Para acceder a los caracteres de una cadena de caracteres, debemos informar el índice —o posición del carácter— entre corchetes (`[]`). Como el primer carácter de una cadena de caracteres es el de índice 0, podemos acceder a valores de 0 hasta el tamaño de la cadena de caracteres menos 1. Luego, si la cadena de caracteres contiene 9 caracteres, podremos acceder a los caracteres de 0 a 8. Vea el resultado de algunas pruebas con cadenas de caracteres en la lista 3.8. Si tratamos de acceder a un índice mayor que la cantidad de caracteres de la cadena de caracteres, el intérprete emitirá un mensaje de error.

► Lista 3.8 – Manipulación de cadenas de caracteres en el intérprete

```
>>> a = "ABCDEF"
>>> print(a[0])
A
>>> print(a[1])
B
>>> print(a[5])
F
>>> print(a[6])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> print(len(a))
6
```

3.4.1 Operaciones con cadenas de caracteres

Las variables de tipo cadena de caracteres soportan operaciones como rebanado, concatenación y composición. Por rebanado, podemos entender la capacidad de utilizar sólo una parte de una cadena de caracteres, o una rebanada. La concatenación es poder juntar dos o más cadenas de caracteres en una nueva cadena de caracteres mayor. La composición es muy utilizada en mensajes que enviamos a la pantalla y consiste en utilizar cadenas de caracteres como modelos donde podemos introducir otras cadenas de caracteres. Veremos cada una de esas operaciones en las secciones a continuación.

3.4.1.1 Concatenación

El contenido de variables cadena de caracteres puede ser sumado, o mejor, concatenado. Para concatenar dos cadenas de caracteres, utilizamos el operador de adición (+). Así, "AB" + "C" es igual a "ABC". Un caso especial de concatenación es la repetición de una cadena de caracteres varias veces. Para eso, utilizamos el operador de multiplicación (*):

"A" * 3 es igual a "AAA". Veamos algunos ejemplos en la lista 3.9.

► Lista 3.9 – Ejemplo de concatenación

```
>>> s = "ABC"
>>> print(s + "C")
ABCC
>>> print(s + "D" * 4)
ABCD DDD
>>> print("X" + "-"*10 + "X")
X-----X
>>> print(s+"x4 = "+s*4)
ABCx4 = ABCABCABCABC
```

3.4.1.2 Composición

Juntar varias cadenas de caracteres para construir un mensaje no siempre es práctico. Por ejemplo, exhibir que "Juan tiene X años", donde X es una variable numérica.

Usando la composición de cadenas de caracteres de Python, podemos escribir de forma simple y clara:

```
"Juan tiene %d años" % X
```

Donde el símbolo de % fue utilizado para indicar la composición de la cadena de caracteres anterior con el contenido de la variable X. El %d dentro de la primera cadena de caracteres es lo que llamamos el marcador de posición. El marcador indica que en aquella posición estaremos colocando un valor entero, de ahí el %d.

Python soporta diversas operaciones con marcadores. Veremos más sobre marcadores en otras partes del libro. La tabla 3.7 presenta los principales tipos de marcadores. Vea que son diferentes, de acuerdo con el tipo de variable que vamos a utilizar.

Tabla 3.7 – Marcadores

Marcador	Tipo
%d	Números enteros
%s	Cadenas de caracteres
%f	Números decimales

Imagine que necesitamos representar un número como 001 o 002, pero que también puede ser algo como 050 o 561. En ese caso, estamos queriendo presentar un número con tres posiciones, completando con ceros a la izquierda si el número es menor. Podemos realizar esa operación utilizando "%03d" % X. Observe que adicionamos 03 entre el % y el d. Si necesita sólo que el número ocupe tres posiciones, pero no desea ceros a la izquierda, basta retirar el cero y utilizar "%3d" % X. Eso es muy importante cuando estamos grabando datos en un archivo o simplemente exhibiendo informaciones en la pantalla. Veamos algunos ejemplos en la lista 3.10.

► Lista 3.10 – Ejemplo de composición con marcadores

```
>>> edad = 22
>>> print("[%d]" % edad)
[22]
>>> print("[%03d]" % edad)
[022]
>>> print("[%3d]" % edad)
[ 22]
>>> print("[% -3d]" % edad)
[22 ]
```

Cuando formateamos números decimales, podemos utilizar dos valores entre el símbolo de % y a letra f. El primero indica el tamaño total en caracteres a reservar; y el segundo, el número de casas

decimales. Así, `%5.2f` dice que estaremos imprimiendo un número decimal utilizando cinco posiciones, siendo que dos son para la parte decimal. Eso es muy interesante para exhibir el resultado de cálculos o representar dinero. Por ejemplo, para exhibir € 5, Puede utilizar `"€%f" % 5`, pero el resultado no es exactamente lo que esperamos, pues normalmente utilizamos sólo dos dígitos después de la coma cuando hablamos de dinero. Veamos algunos ejemplos en la lista 3.11.

► Lista 3.11 – Ejemplos de composición con números decimales

```
>>> print("%5f" % 5)
5.000000
>>> print("%5.2f" % 5)
5.00
>>> print("%10.5f" % 5)
5.00000
```

El poder de la composición realmente aparece cuando necesitamos combinar varios valores en una nueva cadena de caracteres. Imagine que Juan tiene 22 años y sólo € 51,34 en el bolsillo. Para exhibir ese mensaje, podemos utilizar:

```
"%s tiene %d años y sólo €%5.2f en el bolsillo" % ("Juan", 22, 51.34)
```

Python soporta diversas operaciones con marcadores. Veremos más sobre marcadores en otras partes del libro. Cuando tenemos más de un marcador en la cadena de caracteres, estamos obligados a escribir los valores a sustituir entre paréntesis. Ahora, veamos ejemplos con otros tipos, y utilizando más de una variable en la composición, en la lista 3.12.

► Lista 3.12 – Ejemplo de composición de cadena de caracteres

```
>>> nombre = "Juan"
>>> edad = 22
>>> dinero = 51.34
>>> print("%s tiene %d años y €%f en el bolsillo." % (nombre, edad, dinero))
Juan tiene 22 años y €51.340000 en el bolsillo.
>>> print("%12s tiene %3d años y €%5.2f en el bolsillo." % (nombre, edad, dinero))
    Juan tiene   22 años y €51.34 en el bolsillo.
>>> print("%12s tiene %03d años y €%5.2f en el bolsillo." % (nombre, edad, dinero))
    Juan tiene 022 años y €51.34 en el bolsillo.
>>> print("%-12s tiene %-3d años y €%-5.2f en el bolsillo." % (nombre, edad,
dinero))
Juan           tiene 22  años y €51.34 en el bolsillo.
```

3.4.1.3 Rebanado

El rebanado en Python es muy poderoso. Imagine nuestra cadena de caracteres de ejemplo de la figura 3.2. Podemos rebanarla de modo de escribir sólo sus dos primeros caracteres AB utilizando como índice `[0:2]`. El rebanado funciona con la utilización de dos puntos en el índice de la cadena de

caracteres. El número a la izquierda de los dos puntos indica la posición de comienzo de la rebanada; y el de la derecha, del fin. Mientras tanto, es preciso poner atención al final, pues en el ejemplo anterior utilizamos 2; y la C, que es el carácter en la posición 2; no fue incluido. Decimos que eso sucede porque el final de la rebanada no está incluido en la misma, siendo dejado de lado. Entienda [0:2] como la rebanada de caracteres de la posición 0 hasta la posición 2, sin incluirla, o el intervalo cerrado en 0 y abierto en 2.

Veamos otros ejemplos de rebanadas en la lista 3.13.

► Lista 3.13 – Ejemplo de rebanado

```
>>> s="ABCDEFGHI"
>>> print(s[0:2])
AB
>>> print(s[1:2])
B
>>> print(s[2:4])
CD
>>> print(s[0:5])
ABCDE
>>> print(s[1:8])
BCDEFGH
```

Podemos también omitir el número de la izquierda o el de la derecha para representar el comienzo o el final. Así, [:2] indica desde el comienzo hasta el segundo carácter (sin incluirlo), y [1:] indica del carácter de posición 1 hasta el final de la cadena de caracteres. Observe que en ese caso, no necesitamos saber cuántos caracteres contiene la cadena de caracteres.

Si omitimos el comienzo y el fin de la rebanada, estaremos haciendo solo una copia de todos los caracteres de la cadena de caracteres a una nueva cadena de caracteres.

Podemos también utilizar valores negativos para indicar posiciones a partir de la derecha. Así -1 es el último carácter; -2, el penúltimo; y así sucesivamente. Vea el resultado de pruebas con índices negativos en la lista 3.14.

► Lista 3.14 – Ejemplo de rebanado con omisión de valores y con índices negativos

```
>>> s="ABCDEFGHI"
>>> print(s[:2])
AB
>>> print(s[1:])
BCDEFGHI
>>> print(s[0:-2])
ABCDEFG
>>> print(s[:])
ABCDEFGHI
```

```
>>> print(s[-1:])
I
>>> print(s[-5:7])
EFG
>>> print(s[-2:-1])
H
```

Veremos más sobre cadenas de caracteres en Python en el capítulo 7.

3.5 Secuencias y tiempo

Un programa siempre es ejecutado línea por línea por la computadora, realizando las operaciones descritas en el programa una después de la otra. Cuando trabajamos con variables debemos recordar que el contenido de una variable puede cambiar con el tiempo. Eso es porque cada vez que alteramos el valor de una variable, el valor anterior es sustituido por el nuevo.

Observe el programa de la lista 3.15. La variable **deuda** fue utilizada para registrar cuánto estaba debiendo alguien; y la variable **compra** para registrar el valor de nuevas gastos de esa persona. Como somos justos, la persona empezó sin deudas en ❶.

► Lista 3.15 – Ejemplo de secuencia y tiempo

```
deuda = 0 ❶
compra = 100 ❷
deuda = deuda + compra ❸
compra = 200 ❹
deuda = deuda + compra ❺
compra = 300 ❻
deuda = deuda + compra ❼
compra = 0 ❽
print(deuda) ❾
```

En ❷, tenemos la primera compra en el valor de € 100. Mientras tanto, el valor de la deuda continúa siendo 0, pues aún no alteramos su valor de modo de adicionar la compra realizada. Esto es hecho en ❸. Observe que estamos actualizando el valor de la deuda con el valor actual más la compra.

En ❹, registramos una nueva compra en el valor de € 200. En ese punto, la compra sustituye su valor por € 200, y se pierde el valor anterior de € 100. Como ya sumamos el valor anterior en la variable deuda, esa pérdida no representará un problema.

❺ es idéntica a ❸, pero el resultado es bien diferente. En ese momento, compra vale € 200; y deuda, € 100.

En ❻, alteramos el valor de compra nuevamente. Esa vez la compra fue de € 300.

❼ es idéntica a ❸ y ❺, pero su resultado es diferente, pues en ese momento tenemos compra con un valor de € 300; y deuda igual a € 300 (100 + 200), siendo actualizada a € 600 (300 + 300).

En ❽, simplemente decimos que la compra fue 0, representando que la persona no compró nada

más.

⑨ exhibe el contenido de la variable deuda en la pantalla (600).

La figura 3.3 muestra la evolución del contenido de nuestras dos variables en función del tiempo, representado por el número de la línea.

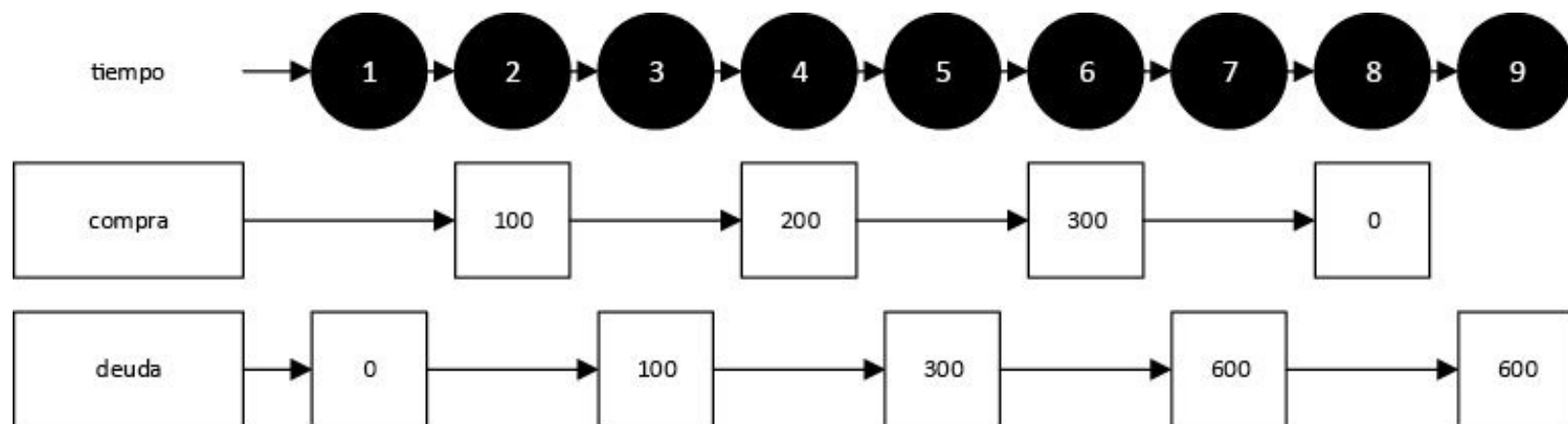


Figura 3.3 – Cambios en el valor de dos variables en el tiempo.

3.6 Rastreo

Una de las principales diferencias entre leer un texto y un programa es justamente seguir los cambios de valores de cada variable a medida que el programa es ejecutado. Entender que el valor de las variables puede cambiar durante la ejecución del programa no es tan sencillo, pero es fundamental para la programación de computadoras. Un programa no puede ser leído como un texto, sino que debe ser cuidadosamente analizado línea por línea. Al escribir sus programas, verifique línea por línea los efectos y cambios causados en el valor de cada variable. Para programar correctamente, usted debe ser capaz de entender qué significa cada línea del programa y los efectos que produce. Esa actividad, llamada de rastreo, es muy importante para entender nuevos programas y para encontrar errores en los programas que usted escribirá.

Para rastrear un programa, utilice lápiz, goma y una hoja de papel. Escriba el nombre de sus variables en la hoja de papel, como si fuesen títulos de columnas, dejando espacio para ser llenados debajo de esos nombres. Lea una línea del programa cada vez y escriba el valor atribuido a cada variable en la otra hoja, en la misma columna en que escribió el nombre de la variable. Si el valor de la variable cambia, escriba el nuevo valor y tache el anterior, uno debajo del otro, formando una columna. Al exhibir algo en la pantalla, escriba también en la otra hoja, como si ella fuese su pantalla (puede dibujar la pantalla como si fuese una variable, pero recuerde dejar un poco más de espacio). En la figura 3.4 se presenta un ejemplo de cómo quedaría el resultado del rastreo del programa de la lista 3.15. El rastreo va a ayudarlo a entender mejor los cambios de valores de sus variables y a acompañar la ejecución del programa, tal como más adelante será hecho por la computadora. Es un proceso detallado que necesita atención. No trate de simplificarlo o de empezar a rastrear en el medio de un programa. Debe rastrear línea por línea, del comienzo al fin del programa. Si Ud. encuentra un error puede parar el rastreo y corregirlo, pero recuerde recomenzar desde el principio siempre que altere el programa o los valores que están siendo rastreados.

Pantalla	deuda	Compra
600	0	100
	100	200
	300	300
	600	0

Figura 3.4 – Ejemplo de rastreo en el papel.

Dominar el rastreo de un programa es esencial para programar y ayuda mucho a entender cómo funcionan realmente los programas. Recuerde que programar es detallar, y que simplemente leer el texto de un programa no es suficiente. Usted debe rastrearlo para entenderlo. Aunque parezca obvio, ese es uno de los errores más comunes cuando se comienza a programar. Si un programa no funciona o si usted no entendió exactamente lo que hace el mismo, el rastreo es la mejor herramienta.

3.7 Entrada de datos

Hasta ahora nuestros programas trabajaron solo con valores conocidos, escritos en el propio programa. Mientras tanto, lo mejor de la programación es poder escribir la solución de un problema y aplicarla varias veces. Para eso necesitamos mejorar nuestros programas de modo de permitir que nuevos valores sean provistos durante su ejecución, para que podamos ejecutarlos con valores diferentes sin alterar los programas en sí.

Llamamos entrada de datos al momento en que el programa recibe datos o valores por un dispositivo de entrada de datos (como el teclado de la computadora) o de un archivo en disco.

La función `input` es utilizada para solicitar datos del usuario. Ella recibe un parámetro, que es el mensaje a ser exhibido, y retorna el valor digitado por el usuario. Veamos un ejemplo en la lista 3.16.

► Lista 3.16 – Entrada de datos

```
x = input("Digite un número: ")
print(x)
```

► Lista 3.17 – Salida en la pantalla, teniendo el 5 como ejemplo de número digitado por el usuario

```
Digite un número: 5
5
```

Veamos otro ejemplo en la lista 3.18.

► Lista 3.18 – Ejemplo de entrada de datos

```
nombre = input("Digite su nombre:") ❶
print("Ud. digitó %s" % nombre)
print("Hola, %s!" % nombre)
```


En ❶, solicitamos la entrada de datos. En este caso el nombre del usuario. El mensaje “Digite su nombre:” es exhibido, y el programa se detiene hasta que el usuario digite ENTER. Sólo entonces el resto del programa es ejecutado. Veamos la salida de datos cuando digitamos “Juan” como nombre en la lista 3.19.

► Lista 3.19 – Resultado de la entrada de datos

Digite su nombre:Juan

Ud. digitó Juan

Hola, Juan!

Ejecute el programa otras veces digitando, por ejemplo, 123 como nombre. Observe que al programa no le importa los valores digitados por el usuario. Esa verificación debe ser hecha por su programa. Veremos cómo hacer eso en otro capítulo, bajo el nombre de validación de datos.

3.7.1 Conversión de la entrada de datos

La función `input` siempre retorna valores del tipo cadena de caracteres, dicho de otro modo; no importa si digitamos solo números, el resultado siempre es una cadena de caracteres. A los efectos de resolver ese pequeño problema, vamos a utilizar la función `int` para convertir el valor retornado en un número entero, y la función `float` para convertirlo en un número decimal o de punto flotante. En la lista 3.20 vemos otro ejemplo usando esas funciones, en el cual debemos calcular el valor de un bono por tiempo de servicio.

► Lista 3.20 – Cálculo de bono por tiempo de servicio

```
años = int(input("Años de servicio: "))
valor_por_año = float(input("Valor por año: "))
bono = años * valor_por_año
print("Bono de € %5.2f" % bono)
```

Veamos, en la pantalla de la lista 3.21, el resultado que obtenemos si ingresamos 10 años y € 25 por año. Observe que escribimos solo 25, y no € 25. Eso es así porque 25 es un número, y € 25 es una cadena de caracteres. Por ahora no vamos a mezclar dos tipos de datos en la misma entrada de datos, para simplificar nuestros programas.

► Lista 3.21 – Resultado del cálculo para 10 años y € 25 por año

Años de servicio: 10

Valor por año: 25

Bono de € 250.00

Ejecute el programa nuevamente con otros valores. Pruebe digitar una letra en años de servicio o en valor por año. Usted recibirá un mensaje de error, pues la conversión de letras en números no es automática. En la próxima sección se explica mejor el problema.

Ejercicio 3.7 Haga un programa que pida dos números enteros. Imprima la suma de esos dos números en la pantalla.

Ejercicio 3.8 Escriba un programa que lea un valor en metros y lo exhiba convertido a milímetros.

Ejercicio 3.9 Escriba un programa que lea la cantidad de días, horas, minutos y segundos del usuario. Calcule el total en segundos.

Ejercicio 3.10 Haga un programa que calcule el aumento de un salario. Debe solicitar el valor del salario y el porcentaje del aumento. Exhiba el valor del aumento y del nuevo salario.

Ejercicio 3.11 Haga un programa que solicite el precio de una mercadería y el porcentaje de descuento. Exhiba el valor del descuento y el precio a pagar.

Ejercicio 3.12 Escriba un programa que calcule el tiempo de un viaje en auto. Pregunte la distancia a recorrer y la velocidad media esperada para el viaje.

Ejercicio 3.13 Escriba un programa que convierta una temperatura digitada en °C a °F. La fórmula para esa conversión es:

$$F = \frac{9 \times C}{5} + 32$$

Ejercicio 3.14 Escriba un programa que pregunte la cantidad de km recorridos por un auto alquilado por el usuario, así como la cantidad de días por los cuales el auto fue alquilado. Calcule el precio a pagar, sabiendo que el auto cuesta € 60 por día y € 0,15 por km recorrido.

Ejercicio 3.15 Escriba un programa para calcular la reducción del tiempo de vida de un fumador. Pregunte la cantidad de cigarrillos fumados por día y cuántos años fumó. Considere que un fumador pierde 10 minutos de vida por cada cigarrillo, calcule cuántos días de vida perderá un fumador. Exhiba el total en días.

3.7.2 Errores comunes

La entrada de datos es un punto sensible en nuestros programas. Como no tenemos manera de prever qué va a digitar el usuario, tenemos que prepararnos para reconocer los errores más comunes. Veamos un programa que lee tres valores en la lista 3.22.

► Lista 3.22 – Entrada de datos con conversión de tipos

```
nombre = input("Digite su nombre: ")
edad = int(input("Digite su edad: "))
saldo = float(input("Digite el saldo de su cuenta bancaria: "))
print(nombre)
```

```
print(edad)
print(saldo)
```

La lista 3.23 muestra el resultado de la ejecución del programa cuando todos los valores son digitados correctamente. “Correcto” significa ausencia de errores durante la función `input` o durante la conversión del valor retornado por la función.

► Lista 3.23 – Ejemplo de entrada de datos

```
Digite su nombre: Juan
Digite su edad: 42
Digite el saldo de su cuenta bancaria: 15756.34
Juan
42
15756.34
```

La lista 3.24 muestra otro ejemplo de entrada de datos exitosa. Observe que, como utilizamos la función `float` para convertir el saldo, aun introduciendo 34 el valor fue convertido a 34.0.

► Lista 3.24 – Ejemplo de entrada de datos

```
Digite su nombre: María
Digite su edad: 28
Digite el saldo de su cuenta bancaria: 34
María
28
34.0
```

Ya en la lista 3.25, tenemos un ejemplo de error durante la entrada de datos. En este caso digitamos letras (abc) que no pueden ser convertidas en un valor entero. Observe que el error interrumpe nuestro programa, mostrando la línea donde ocurrió y el nombre del error. En este caso, el error sucedió en la línea 2 y su nombre es `ValueError: invalid literal es int with base 10: 'abc'`.

► Lista 3.25 – Error de conversión

```
Digite su nombre: Minduim
Digite su edad: abc
Traceback (most recent call last):
  File "input/input2.py", line 2, in <module>
    edad = int(input("Digite su edad: "))
ValueError: invalid literal for int() with base 10: 'abc'
```

La lista 3.26 muestra otro error de conversión, pero esta vez durante la conversión a número decimal, usando la función `float`. La entrada de datos es un poco rústica, deteniéndose en caso de error. Más adelante, aprenderemos sobre excepciones en Python y cómo tratar este tipo de errores. Por ahora basta saber que no estamos validando la entrada, y que nuestros programas aún son

frágiles.

► Lista 3.26 – Error de conversión: letras en lugar de números

Digite su nombre: Juanito

Digite su edad: 31

Digite el saldo de su cuenta bancaria: abc

Traceback (most recent call last):

```
File "input/input2.py", line 3, in <module>
```

```
    saldo = float(input("Digite el saldo de su cuenta bancaria: "))
```

ValueError: could not convert string to float: abc

► Lista 3.27 – Error de conversión: coma en lugar de punto

Digite su nombre: Mary

Digite su edad: 25

Digite el saldo de su cuenta bancaria: 17,4

Traceback (most recent call last):

```
File "input/input2.py", line 3, in <module>
```

```
    saldo = float(input("Digite el saldo de su cuenta bancaria: "))
```

ValueError: invalid literal as float(): 17,4

El error mostrado en la lista 3.27 es muy común en países donde se usa la coma y no el punto como separador entre la parte entera y fraccionaria de un número. En Python, siempre debe digitar valores decimales usando el punto y no la coma, como en español. Así, 17,4 es un valor inválido, pues debería haberse digitado 17.4. Existen recursos en Python para resolver ese tipo de problemas, pero aún es pronto para abordar el asunto.

Condiciones

¿Ejecutar o no ejecutar? Esa es la cuestión...

No siempre todas las líneas de los programas serán ejecutadas. Muchas veces será más interesante decidir qué partes del programa deben ser ejecutadas de acuerdo al resultado de una condición. La base de esas decisiones consistirá en expresiones lógicas que permitan representar elecciones en programas.

4.1 if

Las condiciones sirven para seleccionar cuándo debe ser activada una parte del programa y cuándo debe ser simplemente ignorada. En Python, la estructura de decisión es el `if`. Su formato es presentado en la lista 4.1.

► Lista 4.1 – Formato de la estructura de condicional if

```
if <condición>:  
    bloque verdadero
```

El `if` es nuestro “si”. Podremos entonces entenderlo en español de la siguiente forma: si la condición es verdadera, haga algo.

Veamos un ejemplo presentado en la lista 4.2: leer dos valores e imprimir el mayor de ellos.

► Lista 4.2 – Condiciones

```
a = int(input("Primer valor: "))  
b = int(input("Segundo valor: "))  
if a > b: ❶  
    print("¡El primer número es el mayor!") ❷  
if b > a: ❸  
    print("¡El segundo número es el mayor!") ❹
```

En ❶, tenemos la condición `a > b`. Esa expresión será evaluada, y si su resultado es verdadero, la línea ❷ será ejecutada. Si es falso, la línea ❷ será ignorada. Lo mismo sucede para la condición `b > a` de la línea ❸. Si su resultado es verdadero, la línea ❹ será ejecutada. Si es falso, ignorada.

La secuencia de ejecución del programa es alterada de acuerdo con los valores digitados como el primero y segundo. Digite el programa de la lista 4.2 y ejecútelo dos veces. La primera vez digite un valor mayor primero y uno menor en segundo lugar. La segunda vez invierta esos valores y verifique

si el mensaje en la pantalla también cambió.

Cuando el primer valor es mayor que el segundo, se ejecutan las siguientes líneas: ❶, ❷, ❸. Cuando el primer valor es menor que el segundo, tenemos otra secuencia: ❶, ❸, ❹. Es importante entender que la línea con la condición en sí, es ejecutada aun si el resultado de la expresión es falso.

Las líneas ❶ y ❸ fueron terminadas con el símbolo dos puntos (:). Cuando eso sucede, tenemos el anuncio de un bloque de líneas a continuación. En Python, un bloque se representa desplazando el comienzo de la línea hacia la derecha. El bloque continúa hasta la primera línea que tiene un desplazamiento diferente.

Observe que empezamos a escribir la línea ❷ algunos caracteres más a la derecha que la línea anterior, ❶, que comenzó el bloque. Como la línea ❸ fue escrita más a la izquierda, decimos que el bloque de la línea ❷ ha terminado.

TRIVIA

Python es uno de los pocos lenguajes de programación que utiliza el desplazamiento del texto a la derecha para marcar el comienzo y el fin de un bloque. Otros lenguajes cuentan con palabras especiales para eso, como **BEGIN** y **END**, en Pascal; o las famosas llaves (**{** y **}**), en C y Java.

Ejercicio 4.1 Analice el programa de la lista 4.2. Responda qué sucede si el primero y el segundo valor son iguales. Explique.

Veamos otro ejemplo, donde solicitaremos la edad del auto del usuario, y enseguida escribiremos “nuevo” si el auto tiene menos de tres años; o “viejo” en el caso contrario.

► Lista 4.3 – Auto nuevo o viejo, dependiendo de la edad

```
edad = int(input("Digite la edad de su auto: "))  
if edad <= 3:  
    print("Su auto es nuevo") ❶  
if edad > 3:  
    print("Su auto es viejo") ❷
```

Ejecute el programa de la lista 4.3 y verifique qué aparece en la pantalla. Puede ejecutarlo varias veces con las siguientes edades: 1, 3 y 5. La primera condición es **edad <= 3**. Esa condición decide si la línea con función **print** ❶ será o no ejecutada. Como es una condición simple, podemos ver fácilmente que solo exhibiremos el mensaje “nuevo” para las edades 0, 1, 2 y 3. La segunda condición, **edad > 3**, es la inversa de la primera. Si observa de cerca, no hay un solo número que considere ambas condiciones como verdaderas al mismo tiempo. La segunda decisión es responsable de decidir la impresión del mensaje del auto viejo ❷.

Aunque es obvio que un auto no podría tener valores negativos como edad, el programa no trata ese problema. Vamos alterarlo más adelante para verificar valores inválidos.

Ejercicio 4.2 Escriba un programa que pregunte la velocidad del auto de un usuario. En caso que supere los 80 km/h, exhiba un mensaje diciendo que el usuario fue multado. En este caso, exhiba el valor de la multa, cobrando € 5 por cada km encima de 80 km/h.

Un bloque de líneas en Python puede tener más de una línea, aunque el último ejemplo muestre solo dos bloques con una línea en cada uno. Si necesita dos o más en el mismo bloque, escriba esas líneas en la misma dirección o en la misma columna de la primera línea del bloque. Eso basta para representarlo.

Un problema común aparece cuando tenemos que pagar impuesto de renta. Normalmente pagamos el impuesto de renta por franja de salario. Imagine que para salarios menores de € 1.000,00 no tenemos que pagar impuesto, es decir que tenemos una alícuota de 0%. Para salarios entre € 1.000,00 y € 3.000,00 pagaríamos 20%. Encima de esos valores, la alícuota sería de 35%. Ese problema se parecería mucho al anterior si el impuesto no fuese cobrado de manera diferente para cada franja; quien gana € 4.000,00 tiene los primeros € 1.000,00 exentos de impuesto; para los valores entre € 1.000,00 y € 3.000,00 paga 20%, y para los valores restantes paga 35%. Veamos la solución en la lista del programa 4.4.

► Lista 4.4 – Cálculo del impuesto de renta

```
salario = float(input("Digite el salario para cálculo del impuesto: "))
base = salario ❶
impuesto = 0
if base > 3000: ❷
    impuesto = impuesto + ((base - 3000) * 0.35) ❸
    base = 3000 ❹
if base > 1000: ❺
    impuesto = impuesto + ((base - 1000) * 0.20) ❻
print("Salario: €%6.2f Impuesto a pagar: €%6.2f" % (salario, impuesto))
```

El programa de la lista 4.4 es muy interesante. Trate de ejecutarlo algunas veces y compare el valor impreso con el valor calculado por Ud. Rastree el programa y trate de entender qué hace antes de leer el párrafo siguiente. Verifique qué sucede para salarios de € 500,00, € 1.000,00 y € 1,500,00.

En ❶ tenemos la variable **base** recibiendo una copia de **salario**. Esto es necesario porque, cuando atribuimos un nuevo valor a una variable, el valor anterior es sustituido (y perdido si no lo guardamos en otro lugar). Como vamos a utilizar el valor del salario digitado para exhibirlo en la pantalla, no podemos perderlo; por eso, la necesidad de una variable auxiliar llamada aquí **base**.

En ❷ verificamos si el valor de la variable **base** es mayor que € 3.000,00. Si esto es verdadero, ejecutamos las líneas ❸ y ❹. En ❸, calculamos 35% del valor superior a € 3.000,00. El resultado es almacenado en la variable **impuesto**. Como esa variable contiene el valor a pagar para esa cantidad, actualizaremos el valor de **base** para € 3.000,00 ❹, pues el que supera ese valor ya fue tarifado.

En ❺ verificamos si el valor de base es mayor que € 1.000,00, calculando 20% de impuesto en ❻, en caso que sea verdadero.

Veamos el rastreo para un salario de € 500,00:

salario	base	impuesto
500	500	0

Para un salario de € 1.500,00:

salario	base	impuesto
1500	1500	0
		100

Para un salario de € 3.000,00:

salario	base	impuesto
3000	3000	0
		400

Para un salario de € 5.000,00:

salario	base	impuesto
5000	5000	0
	3000	700
		1100

Ejercicio 4.3 Escriba un programa que lea tres números y que imprima el mayor y el menor.

Ejercicio 4.4 Escriba un programa que pregunte el salario del empleado y calcule el valor del aumento. Para salarios superiores a € 1.250,00, calcule un aumento de 10%. Para los inferiores o iguales, de 15%.

4.2 else

Cuando hay problemas, como en el caso del mensaje del auto viejo (Lista 4.3), donde la segunda condición es simplemente la inversa de la primera, podemos usar otra forma de **if** para simplificar los programas. Esa forma es la cláusula **else**, para especificar qué hacer en caso que el resultado de la evaluación de la condición sea falso, si necesitamos un nuevo **if**. Veamos cómo quedaría el programa reescrito para usar **else** en la lista 4.5.

► Lista 4.5 – Auto nuevo o viejo, dependiendo de la edad con else

```
edad = int(input("Digite la edad de su auto: "))
if edad <= 3:
    print("Su auto es nuevo")
else: ❶
    print("Su auto es viejo") ❷
```

Vea que en ❶ utilizamos “:” después de **else**. Esto es necesario porque **else** empieza un bloque, del mismo modo que **if**. Es importante notar que debemos escribir **else** en la misma columna del **if**, o sea, con el mismo margen o retiro. Así, el intérprete reconoce que **else** se refiere a un determinado **if**. Ud. obtendrá un error en caso que no alinee esas dos estructuras en la misma columna.

La ventaja de usar **else** es dejar los programas más claros, una vez que podemos expresar qué hacer en caso que la condición especificada en **if** sea falsa. La línea ❷ sólo es ejecutada si la condición

edad <= 3 es falsa.

Ejercicio 4.5 Ejecute el programa (Lista 4.5) y pruebe algunos valores. Verifique si los resultados fueron los mismos del programa anterior (Lista 4.3).

Ejercicio 4.6 Escriba un programa que pregunte la distancia que un pasajero desea recorrer en kms. Calcule el precio del pasaje, cobrando € 0,50 por km para viajes de hasta de 200 km, y € 0,45 para viajes más largos.

4.3 Estructuras anidadas

No siempre nuestros programas serán tan simples. Muchas veces necesitaremos anidar varios `if` para obtener el comportamiento deseado del programa. Anidar, en este caso, es utilizar un `if` dentro de otro.

Veamos el ejemplo de calcular la cuenta de un teléfono celular de la empresa Chau. Los planes de la empresa Chau son muy interesantes y ofrecen precios diferenciados de acuerdo con la cantidad de minutos usados por mes. Abajo de 200 minutos la empresa cobra € 0,20 por minuto. Entre 200 y 400 minutos, el precio es de € 0,18. Encima de 400 minutos, el precio por minuto es de € 0,15. El programa de la lista 4.6 resuelve ese problema.

► **Lista 4.6 – Cuenta de teléfono con tres franjas de precio**

```
minutos=int(input("Cuántos minutos Ud. utilizó este mes:"))
if minutos < 200: ❶
    precio = 0.20 ❷
else:
    if minutos < 400: ❸
        precio = 0.18 ❹
    else: ❺
        precio = 0.15 ❻
print("Ud. va pagar este mes: €%6.2f" % (minutos * precio))
```

En ❶, tenemos la primera condición: `minutos < 200`. Si la cantidad de minutos es menor que 200, atribuimos 0,20 al precio en ❷. Hasta aquí, nada nuevo. Observe que el `if` de ❸ está dentro del `else` de la línea anterior: decimos que está anidado dentro de `else`. La condición de ❸, `minutos < 400`, decide si vamos ejecutar la línea de ❹ o a de ❻. Observe que el `else` de ❺ está alineado con el `if` de ❸. Al final, calculamos e imprimimos el precio en la pantalla. Recuerde que la alineación del texto es muy importante en Python.

Veamos, por ejemplo, la situación en que cinco categorías son necesarias. Hagamos un programa que lea la categoría de un producto y determine el precio por la tabla 4.1.

Tabla 4.1 – Categorías de producto y precio

Categoría	Precio
1	10,00

2	18,00
3	23,00
4	26,00
5	31,00

A la izquierda de la lista 4.7, encontrará los números de línea del programa, numeradas de 1 a 19. Esos números sirven solo para ayudar al entendimiento de la explicación que sigue: recuerde no digitarlos.

► Lista 4.7 – Categoría x precio

```

1 categoría = int(input("Digite la categoría del producto:"))
2 if categoría == 1:
3     precio = 10
4 else:
5     if categoría == 2:
6         precio = 18
7     else:
8         if categoría == 3:
9             precio = 23
10        else:
11            if categoría == 4:
12                precio = 26
13            else:
14                if categoría == 5:
15                    precio = 31
16                else:
17                    print("¡Categoría inválida, digite un valor entre 1 y 5!")
18                    precio = 0
19 print("El precio del producto es: €%6.2f" % precio)

```

Observe que la alineación se volvió un gran problema, una vez que tuvimos que desplazar a la derecha cada **else**.

En el programa de la lista 4.7, introdujimos el concepto de validación de la entrada. Esta vez, si el usuario digita un valor inválido, recibirá un mensaje de error en la pantalla. Nada muy práctico o bonito.

Veamos la ejecución de las líneas, dependiendo de la categoría digitada en la tabla 4.2.

Tabla 4.2 – Líneas ejecutadas

Categoría	Líneas ejecutadas
1	1,2,3,19
2	1,2,4,5,6,19
3	1,2,4,5,7,8,9,19
4	1,2,4,5,7,8,10,11,12,19

5	1,2,4,5,7,8,10,11,13,14,15,19
otras	1,2,4,5,7,8,10,11,13,14,16,17,18,19

Cuando leemos un programa con estructuras anidadas, debemos prestar mucha atención para visualizar correctamente los bloques. Observe cómo la alineación es importante.

Ejercicio 4.7 Rastree el programa de la lista 4.7. Compare su resultado con el presentado en la tabla 4.2.

4.4 elif

Python presenta una solución muy interesante al problema de múltiples `ifs` anidados. La cláusula `elif` sustituye un par `else if`, pero sin crear otro nivel de estructura, evitando problemas de desplazamientos innecesarios a la derecha.

Vamos a volver a considerar el problema de la lista 4.7, esta vez usando `elif`. Vea el resultado en el programa de la lista 4.8.

► Lista 4.8 – Categoría x precio, usando elif

```

categoría = int(input("Digite la categoría del producto:"))
if categoría == 1:
    precio = 10
elif categoría == 2:
    precio = 18
elif categoría == 3:
    precio = 23
elif categoría == 4:
    precio = 26
elif categoría == 5:
    precio = 31
else:
    print("¡Categoría inválida, digite un valor entre 1 y 5!")
    precio = 0
print("El precio del producto es: €%6.2f" % precio)

```

Ejercicio 4.8 Escriba un programa que lea dos números y que pregunte qué operación desea realizar. Usted debe poder calcular la suma (+), sustracción (-), multiplicación (*) y división (/). Exhiba el resultado de la operación solicitada.

Ejercicio 4.9 Escriba un programa para aprobar el préstamo bancario para la compra de una casa. El programa debe preguntar el valor de la casa a comprar, el salario y la cantidad de años a pagar. El valor de la cuota mensual no puede ser superior a 30% del salario. Calcule el valor de la cuota dividiendo el valor de la casa a comprar por el número de meses a pagar.

Ejercicio 4.10 Escriba un programa que calcule el precio a pagar por el suministro de energía eléctrica. Pregunte la cantidad de kwh consumida y el tipo de instalación: R para residencias, I para industrias y C para comercios. Calcule el precio a pagar de acuerdo con la siguiente tabla.

Precio por tipo y rango de consumo		
Tipo	Faixa (kWh)	Precio
Residencial	hasta 500	€ 0,40
	más de 500	€ 0,65
Comercial	hasta 1000	€ 0,55
	más de 1000	€ 0,60
Industrial	hasta 5000	€ 0,55
	más de 5000	€ 0,60

Repeticiones

Las repeticiones representan la base de varios programas. Son utilizadas para ejecutar la misma parte de un programa varias veces, normalmente dependiendo de una condición. Por ejemplo, para imprimir tres números en la pantalla, podríamos escribir un programa como el presentado en la lista 5.1.

► Lista 5.1 – Imprimiendo de 1 a 3

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

Podemos imaginar que para imprimir tres números, comenzando del 1 hasta el 3, debemos variar `print(x)`, donde `x` varía de 1 a 3. Veamos otra solución para el problema en la lista 5.2.

► Lista 5.2 – Imprimiendo de 1 a 3, usando una variable

```
x=1
print(x)
x=2
print(x)
x=3
print(x)
```

Otra solución sería incrementar el valor de `x` después de cada `print`. Veamos esa solución en la lista 5.3.

► Lista 5.3 – Imprimiendo de 1 a 3, incrementando

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

Sin embargo, si el objetivo fuese escribir 100 números, la solución no sería tan agradable, ¡Pues tendríamos que escribir por lo menos 200 líneas! La estructura de repetición aparece para ayudarnos a resolver ese tipo de problema.

Una de las estructuras de repetición de Python es el **while**, que repite un bloque mientras la condición sea verdadera. Su formato se presenta en la lista 5.4, donde *condición* es una expresión lógica, y *bloque* representa las líneas de programa a repetir mientras el resultado de la condición sea verdadero.

► Lista 5.4 – Formato de la estructura de repetición con while

```
while <condición>:  
    bloque
```

Para resolver el problema de escribir tres números utilizando el **while**, escribiríamos un programa como el de la lista 5.5.

► Lista 5.5 – Imprimiendo de 1 a 3 con while

```
x=1 ❶  
while x <= 3: ❷  
    print(x) ❸  
    x = x + 1 ❹
```

La ejecución de ese programa sería un poco diferente del que vimos hasta ahora. Primero, ❶ sería ejecutado inicializando la variable **x** con el valor 1. La línea ❷ sería una combinación de estructura condicional con estructura de repetición. Podemos entender la condición del **while** del mismo modo que la condición de **if**. La diferencia está en que si la condición es verdadera, repetiremos las líneas ❸ y ❹ (bloque) mientras la evaluación de la condición sea verdadera.

En ❸ tendremos la impresión en la pantalla propiamente dicha, donde **x** es 1. En ❹ tenemos que el valor de **x** es incrementado en 1. Como **x** vale 1, **x + 1** valdrá 2. Ese nuevo valor es entonces atribuido a **x**. La parte nueva es que la ejecución no termina después de ❹ que es el fin del bloque, sino que retorna a ❷. Es ese retorno lo que hace especial la estructura de repetición.

Ahora, **x** vale 2 y **x <= 3** continúa siendo verdadero (**True**), luego, el bloque será ejecutado otra vez. ❸ realizará la impresión del valor 2, y ❹ actualizará el valor de **x** para **x + 1**; en este caso, **2 + 1 = 3**. La ejecución vuelve nuevamente a la línea ❷.

La condición en ❷ es evaluada, y como **x** vale 3, y **x <= 3** continúa siendo verdadera, las líneas ❸ y ❹ son ejecutadas, exhibiendo 3 y actualizando el valor de **x** a 4 (**3 + 1**).

En ese punto, ❷, tenemos que **x** vale 4 y que la condición **x <= 3** resulta Falsa (**False**), terminando, así, la repetición del bloque.

Ejercicio 5.1 Modifique el programa para exhibir los números de 1 a 100.

Ejercicio 5.2 Modifique el programa para exhibir los números de 50 a 100.

Ejercicio 5.3 Haga un programa para escribir la cuenta regresiva del lanzamiento de un cohete. El programa debe imprimir 10, 9, 8, ..., 1, 0 y ¡Fuego! en la pantalla.

5.1 Contadores

El poder de las estructuras de repetición es muy interesante, principalmente cuando utilizamos condiciones con más de una variable. Imagine un problema donde deberíamos imprimir los números enteros entre 1 y un valor digitado por el usuario. Vamos a modificar el programa de la lista 5.5 de modo que el último número a imprimir sea informado por el usuario. El programa ya modificado se presenta en la lista 5.6.

► Lista 5.6 – Impresión de 1 hasta un número digitado por el usuario

```
fin = int(input("Digite el último número a imprimir:")) ❶
x = 1
while x <= fin: ❷
    print(x) ❸
    x = x + 1 ❹
```

En este caso, el programa imprimirá desde 1 hasta el valor digitado en ❶. En ❷ utilizamos la variable `fin` para representar el límite de nuestra repetición.

Ahora vamos a analizar lo que realizamos con la variable `x` dentro de la repetición. En ❸, simplemente se imprime el valor de `x`. En ❹ actualizamos el valor de `x` con `x + 1`, o sea, con el próximo valor entero. Cuando realizamos ese tipo de operación dentro de una repetición estamos contando. Luego, diremos que `x` es un contador. Un contador es una variable utilizada para contar el número de ocurrencias de un determinado evento; en este caso, el número de repeticiones del `while`, que satisface las necesidades de nuestro problema.

Pruebe ese programa con varios valores, primero digitando 5, después 500, y por fin 0 (cero). Probablemente 5 y 500 producirán los resultados esperados, o sea, la impresión de 1 hasta 5, o de 1 hasta 500. Sin embargo, cuando digitamos cero, nada sucede, y el programa termina enseguida, sin impresión.

Analizando nuestro programa, vemos que cuando la variable `fin` vale 0—o sea, cuando digitamos 0 en ❶—, tenemos que la condición en ❷ es `x <= fin`. Como `x` es 1 y `fin` es 0, tenemos que `1 <= 0` es falso desde la primera ejecución, lo cual hace que el bloque a repetir no sea ejecutado, una vez que su condición de entrada es falsa. Lo mismo sucedería con la inserción de valores negativos.

Imagine que el problema ahora sea un poco diferente: imprimir solo los números pares entre 0 y un número digitado por el usuario, de forma bien similar al problema anterior. Podríamos resolver el problema con un `if` para probar si `x` es par o impar antes de imprimir. Vale recordar que un número es par cuando es 0 o múltiplo de 2. Cuando es múltiplo de 2, tenemos que el resto de la división de ese número por 2 es 0; dicho de otro modo, el resultado es una división exacta, sin resto. En Python podemos escribir ese test como `x % 2 == 0` (resto de la división de `x` por 2 es igual a cero), alterando el programa anterior y obteniendo el de la lista 5.7.

► Lista 5.7 – Impresión de números pares de 0 hasta un número digitado por el usuario

```
fin=int(input("Digite el último número a imprimir:"))
x = 0 ❶
while x <= fin:
```

```
if x % 2 == 0: ❷
    print(x) ❸
x = x + 1
```

Vea que, para empezar a imprimir desde el 0, y no desde 1, modificamos ❶. Un detalle importante es que ❸ es un bloque dentro de **if** ❷, siendo para eso desplazado a la derecha. Ejecute el programa y verifique su resultado.

Ahora, finalmente, estamos resolviendo el problema; pero podríamos resolverlo de forma aun más simple si adicionásemos 2 a **x** en cada repetición. Eso garantizaría que **x** siempre fuese par. Veamos el programa de la lista 5.8.

► Lista 5.8 – Impresión de números pares de 0 hasta un número digitado por el usuario, sin if

```
fin = int(input("Digite el último número a imprimir:"))
x = 0
while x <= fin:
    print(x)
    x = x + 2
```

Esos dos ejemplos muestran que existe más de una solución para el problema, que podemos escribir programas diferentes y obtener la misma solución. Esas soluciones pueden ser a veces más complicadas, a veces más simples, pero aun así correctas.

Ejercicio 5.4 Modifique el programa anterior para imprimir de 1 hasta el número digitado por el usuario, pero, esta vez, imprima solo números impares.

Ejercicio 5.5 Reescriba el programa anterior para escribir los 10 primeros múltiplos de 3.

Veamos otro tipo de problema. Imagine que tiene que imprimir la tabla de sumar de un número digitado por el usuario. Esa tabla debe ser impresa de 1 a 10, siendo **n** el número digitado por el usuario. Tendríamos, así, **n+1**, **n+2**, ... **n+10**. Verifique la solución en la lista 5.9.

► Lista 5.9 – Tabla simple

```
n = int(input("Tabla de:"))
x = 1
while x <= 10:
    print(n + x)
    x = x + 1
```

Ejecute el programa anterior y pruebe diversos valores.

Ejercicio 5.6 Altere el programa anterior para exhibir los resultados en el mismo formato de una tabla: 2 x 1 = 2, 2 x 2=4, ...

Ejercicio 5.7 Modifique el programa anterior de modo que el usuario también digite el comienzo y el fin de la tabla, en vez de empezar con 1 y terminar con 10.

Ejercicio 5.8 Escriba un programa que lea dos números. Imprima el resultado de la multiplicación del primero por el segundo. Utilice solo los operadores de suma y sustracción para calcular el resultado. Recuerde que podemos entender la multiplicación de dos números como sumas sucesivas de uno de ellos. Así, $4 \times 5 = 5 + 5 + 5 + 5 = 4 + 4 + 4 + 4 + 4$.

Ejercicio 5.9 Escriba un programa que lea dos números. Imprima la división entera del primero por el segundo, así como el resto de la división. Utilice solo los operadores de suma y sustracción para calcular el resultado. Recuerde que podemos entender el cociente de la división de dos números como la cantidad de veces que podemos retirar el divisor del dividendo. Logo, $20 \div 4 = 5$, una vez que podemos sustraer 4 cinco veces de 20.

Los contadores también pueden ser útiles cuando son usados con condiciones dentro de los programas. Veamos un programa para corregir un test de múltiple opción con tres preguntas. La respuesta de la primera es “b”; la de la segunda, “a”; y la de la tercera, “d”. El programa de la lista 5.10 cuenta un punto por cada respuesta correcta.

► Lista 5.10 – Recuento de preguntas correctas

```
puntos = 0
pregunta = 1
while pregunta <= 3:
    respuesta = input("Respuesta de la pregunta %d: " % pregunta)
    if pregunta == 1 and respuesta == "b":
        puntos = puntos + 1
    if pregunta == 2 and respuesta == "a":
        puntos = puntos + 1
    if pregunta == 3 and respuesta == "d":
        puntos = puntos + 1
    pregunta += 1
print("El alumno hizo %d punto(s)" % puntos)
```

Ejecute el programa y digite todas las respuestas correctas, después trate con respuestas diferentes. Vea que estamos verificando solo respuestas simples de una sola letra y que consideramos solo letras minúsculas. En Python, una letra minúscula es diferente de una mayúscula. Si usted digita “A” en la segunda pregunta en vez de “a”, el programa no considerará esa respuesta correcta. Una solución para ese tipo de problema es utilizar el operador lógico **or** y verificar la respuesta mayúscula y minúscula. Por ejemplo, `pregunta == 1 and (respuesta == "b" or respuesta == "B")`.

Ejercicio 5.10 Modifique el programa de la lista 5.10 para que acepte respuestas con letras mayúsculas y minúsculas en todas las preguntas.

Aunque esa verificación resuelva el problema, veremos que si digitamos un espacio en blanco, antes

o después de la respuesta, también será considerada equivocada. Siempre que trabajamos con cadenas de caracteres, ese tipo de problemas debe ser controlado. Veremos más sobre el asunto en el capítulo 7.

5.2 Acumuladores

No solo necesitamos contadores. En programas para calcular el total de una suma, por ejemplo, necesitaremos acumuladores. La diferencia entre un contador y un acumulador es que en los contadores el valor adicionado es constante, y en los acumuladores es variable. Veamos un programa que calcule la suma de 10 números, en la lista 5.11. En este caso, **suma** ❶ es un acumulador y **n** ❷ es un contador.

► Lista 5.11 – Suma de 10 números

```
n = 1
suma = 0
while n <= 10:
    x = int(input("Digite el %d número:%n"))
    suma = suma + x ❶
    n = n + 1 ❷
print("Suma: %d" % suma)
```

Podemos definir la media aritmética como la suma de varios números divididos por la cantidad de números sumados. Así, si sumamos tres números, 4, 5 y 6, tendríamos la media aritmética como $(4+5+6) / 3$, donde 3 es la cantidad de números. Si llamamos al primer número n_1 , al segundo n_2 , y al tercero n_3 , tendremos $(n_1 + n_2 + n_3) / 3$.

Veamos, en la lista 5.12, un programa que calcula la media de cinco números digitados por el usuario. Si llamamos al primer valor digitado n_1 , al segundo n_2 , y así sucesivamente, tendremos que:

$$media = (n_1 + n_2 + n_3 + n_4 + n_5) / 5 = \frac{n_1 + n_2 + n_3 + n_4 + n_5}{5}$$

En vez de utilizar cinco variables, vamos a acumular los valores a medida que son leídos.

► Lista 5.12 – Cálculo de media con acumulador

```
x = 1
suma = 0 ❶
while x <= 5:
    n = int(input("%d Digite el número:" % x))
    suma = suma + n ❷
    x = x + 1
print("Media: %5.2f" % (suma / 5)) ❸
```

En este caso, tenemos que **x** es un contador y **n** el valor digitado por el usuario. La variable **suma** es creada en ❶ e inicializada con 0. A diferencia de **x**, que recibe 1 en cada pasaje, la variable **suma**, en

❷, es incrementada por el valor digitado por el usuario. Podemos decir que el incremento de **suma** no es un valor constante, pues varía con el valor digitado por el usuario. Podemos también decir que **suma** acumula los valores de **n** en cada repetición. Luego, diremos que la variable **suma** es un acumulador.

Los acumuladores son muy interesantes cuando no sabemos o no conseguimos obtener el total de la suma por la simple multiplicación de dos números. En el caso del cálculo de la media, el valor de **n** puede ser diferente cada vez que el usuario digite un valor.

Ejercicio 5.11 Escriba un programa que pregunte el depósito inicial y la tasa de intereses de un ahorro. Exhiba los valores mes por mes para los 24 primeros meses. Escriba el lucro total por intereses en el período.

Ejercicio 5.12 Altere el programa anterior de modo de preguntar también el valor depositado mensualmente. Ese valor será depositado al comienzo de cada mes, y usted debe considerarlo para el cálculo de intereses del mes siguiente.

Ejercicio 5.13 Escriba un programa que pregunte el valor inicial de una deuda y el interés mensual. Pregunte también el valor mensual que será pagado. Imprima el número de meses para que la deuda sea pagada, el total pagado y el total de intereses pagado.

5.3 Interrumpiendo la repetición

Aunque es muy útil, la estructura **while** solo verifica su condición de detención al comienzo de cada repetición. Dependiendo del problema, la habilidad de terminar **while** dentro del bloque a repetir puede ser interesante.

La instrucción **break** es utilizada para interrumpir la ejecución de **while** independientemente del valor actual de su condición. Veamos el ejemplo de la lectura de valores hasta que digitemos 0 (cero) en el programa de la lista 5.13.

► Lista 5.13 – Interrumpiendo la repetición

```
s = 0
while True: ❶
    v = int(input("Digite un número a sumar o 0 para salir:"))
    if v == 0:
        break ❷
    s = s + v ❸
print(s) ❹
```

En ese ejemplo, sustituimos la condición del **while** por **True** en ❶. De esa forma, el **while** se ejecutará indefinidamente, pues el valor de su condición de parada (**True**) es constante. En ❷ tenemos la instrucción **break** que se activa dentro de un **if**, específicamente cuando **v** es cero. Sin embargo, ahora **v** es diferente de cero, por lo cual la repetición continuará sumando **v** a **s** en ❸. Cuando **v** es igual a cero (0), tendremos que ❷ sigue siendo ejecutada, terminando la repetición y

transfiriendo la ejecución a ❹, que, entonces, exhibe el valor de `s` en la pantalla.

Ejercicio 5.14 Escriba un programa que lea números enteros del teclado. El programa debe leer los números hasta que el usuario digite 0 (cero). Al final de la ejecución, exhiba la cantidad de números digitados, así como la suma y la media aritmética.

Ejercicio 5.15 Escriba un programa para controlar una pequeña máquina registradora. Usted debe solicitarle al usuario que digite el código del producto y la cantidad comprada. Utilice la tabla de códigos que aparece abajo para obtener el precio de cada producto:

Código	Precio
1	0,50
2	1,00
3	4,00
5	7,00
9	8,00

Su programa debe exhibir el total de las compras después que el usuario digite 0. Cualquier otro código debe generar el mensaje de error “Código inválido”.

Veamos como ejemplo un programa que lea un valor y que imprima la cantidad de billetes necesarios para pagar ese mismo valor, presentado en la lista 5.14. Para simplificar, vamos a trabajar solo con valores enteros y con billetes de € 50, € 20, € 10, € 5 y € 1.

Ejercicio 5.16 Ejecute el programa (Lista 5.14) para los siguientes valores: 501, 745, 384, 2, 7 y 1.

Ejercicio 5.17 ¿Qué sucede si digitamos 0 (cero) en el valor a pagar?

Ejercicio 5.18 Modifique el programa para trabajar también con billetes de € 100.

Ejercicio 5.19 Modifique el programa para aceptar valores decimales, o sea, para que cuente monedas de 0,01, 0,02, 0,05, 0,10 y 0,50.

Ejercicio 5.20 ¿Qué sucede si digitamos 0,001 en el programa anterior? En caso que no funcione, altérelo de modo de corregir el problema.

► **Lista 5.14 – Recuento de billetes**

```
valor = int(input("Digite el valor a pagar:"))
billetes = 0
actual = 50
apagar = valor
while True:
    if actual <= apagar:
```

```

    apagar -= actual
    billetes += 1
else:
    print("%d billete(s) de €%d" % (billetes, actual))
    if apagar == 0:
        break
    if actual == 50:
        actual = 20
    elif actual == 20:
        actual = 10
    elif actual == 10:
        actual = 5
    elif actual == 5:
        actual = 1
    billetes = 0

```

5.4 Repeticiones anidadas

Podemos combinar varios `while` de modo de obtener resultados más interesantes, como la repetición con incremento de dos variables. Imagine que tiene que imprimir las tablas de multiplicación de 1 a 10. Veamos cómo hacer eso, leyendo la lista del programa 5.15.

► Lista 5.15 – Impresión de tablas

```

tabla = 1
while tabla <= 10: ❶
    número = 1 ❷
    while número <= 10: ❸
        print("%d x %d = %d" % (tabla, número, tabla * número))
        número += 1 ❹
    tabla += 1 ❺

```

En ❶ tenemos nuestro primer `while`, creado para repetir su bloque, ahora el valor de `tabla` es menor o igual a 10. En ❷ tenemos la inicialización de la variable `número` dentro del primer `while`. Esto es importante porque necesitamos volver a multiplicar por 1 cada nuevo valor de la variable `tabla`. Finalmente, en ❸ tenemos el segundo `while` con la condición de parada `número <= 10`. Ese `while` ejecutará sus repeticiones dentro del primero, o sea, el punto de ejecución pasa de ❹ para ❸; ahora la condición es verdadera. Vea que en ❹ utilizamos el operador `+=` para representar `número = número + 1`. Cuando `número` valga 11, la condición en ❸ resultará falsa, y la ejecución del programa continuará a partir de la línea ❺. En ❺ incrementamos el valor de `tabla` y volveremos a ❶, donde será verificada la condición del primer `while`. Como resulta verdadero, volveremos a ejecutar ❷, reiniciando la variable `número` con el valor 1. ❷ es muy importante para que podamos nuevamente ejecutar el segundo `while`, responsable de imprimir la tabla en la pantalla.

Veamos el mismo problema, pero sin utilizar repeticiones anidadas, como se presenta la lista 5.16.

► Lista 5.16 – Impresión de tablas sin repeticiones anidadas

```
tabla = 1
número = 1
while tabla <= 10:
    print("%d x %d = %d" % (tabla, número, tabla * número))
    número += 1
    if número == 11:
        número = 1
        tabla += 1
```

Ejercicio 5.21 Reescriba el programa de la lista 5.14 de tal modo que continúe ejecutándolo hasta que el valor digitado sea 0. Utilice repeticiones anidadas.

Ejercicio 5.22 Escriba un programa que exhiba una lista de opciones (menú): adición, sustracción, división, multiplicación y salir. Imprima la tabla de la operación elegida. Repita la ejecución del programa hasta que la opción salida sea elegida.

Ejercicio 5.23 Escriba un programa que lea un número y verifique si es o no un número primo. Para hacer esa verificación, calcule el resto de la división del número por 2 y después por todos los números impares hasta el número leído. Si el resto de una de esas divisiones es igual a cero, el número no es primo. Observe que 0 y 1 no son primos y que 2 es el único número primo que es par.

Ejercicio 5.24 Modifique el programa anterior de modo tal de leer un número n . Imprima los n primeros números primos.

Ejercicio 5.25 Escriba un programa que calcule la raíz cuadrada de un número. Utilice el método de Newton para obtener un resultado aproximado. Siendo n el número del cual obtener la raíz cuadrada, considere la base $b=2$. Calcule p usando la fórmula $p=(b+(n/b))/2$. Ahora calcule el cuadrado de p . A cada paso, haga $b=p$ y recalculé p usando la fórmula presentada. Pare cuando la diferencia absoluta entre n y el cuadrado de p sea menor que 0,0001.

Ejercicio 5.26 Escriba un programa que calcule el resto de la división entera entre dos números. Utilice solo las operaciones de suma y sustracción para calcular el resultado.

Ejercicio 5.27 Escriba un programa que verifique si un número es palíndromo. Un número es palíndromo si continúa igual en caso que sus dígitos sean invertidos. Ejemplos: 454, 10501

Listas

Las listas son un tipo de variable que permite el almacenamiento de varios valores, a los que se accede a través de un índice. Una lista puede contener cero o más elementos de un mismo tipo o de tipos diversos, pudiendo inclusive contener otras listas. El tamaño de una lista es igual a la cantidad de elementos que ella contiene.

Podemos imaginar una lista como un edificio de apartamentos donde la planta baja es el piso cero, el primer piso es el piso 1 y así sucesivamente. El índice es utilizado para especificar el “apartamento” donde guardaremos nuestros datos.

En un edificio de seis pisos, tendremos números de piso que variarán entre 0 y 5. Si llamamos **P** a nuestro edificio, tendremos **P[0]** como la dirección de la planta baja, **P[1]** como la dirección del primer piso, continuando así hasta **P[5]**. En Python, **P** sería el nombre de la lista; y el número entre corchetes el índice.

Las listas son más flexibles que los edificios y pueden crecer o disminuir con el tiempo. Veamos cómo crear una lista en Python en la lista 6.1.

► Lista 6.1 – Una lista vacía

```
L=[]
```

Esa línea crea una lista llamada **L**, con cero elementos; o sea, una lista vacía. Los corchetes (`[]`) después del símbolo de igualdad sirven para indicar que **L** es una lista. Veamos ahora cómo crear una lista **Z**, con 3 elementos, en la lista 6.2.

► Lista 6.2 – Una lista con tres elementos

```
Z=[15, 8, 9]
```

La lista **Z** fue creada con tres elementos: 15, 8 y 9. Decimos que el tamaño de la lista **Z** es 3. Como el primer elemento tiene índice 0, tenemos que el último elemento es **Z[2]**. Vea el resultado de pruebas con **Z** en la lista 6.3.

► Lista 6.3 – Acceso a una lista

```
>>> Z=[15, 8, 9]
```

```
>>> Z[0]
```

```
15
```

```
>>> Z[1]
```

```
8
```

```
>>> Z[2]
```

Utilizando el nombre de la lista y un índice, podemos cambiar el contenido de un elemento. Observe las pruebas en el intérprete, presentadas en la lista 6.4.

► Lista 6.4 – Modificación de una lista

```
>>> Z=[15, 8, 9]
>>> Z[0]
15
>>> Z[0]=7
>>> Z[0]
7
>>> Z
[7, 8, 9]
```

Cuando creamos la lista Z, el primer elemento era el número 15. Por eso, Z[0] era 15. Cuando ejecutamos Z[0]=7, alteramos el contenido del primer elemento, que pasó a ser 7. Eso puede ser verificado cuando pedimos que se imprima Z, ahora con 7, 8 y 9 como elementos.

Veamos un ejemplo donde un alumno tiene cinco notas y en el cual deseamos calcular su media aritmética. Vea el programa en la lista 6.5.

► Lista 6.5 – Cálculo de la media

```
notas = [6, 7, 5, 8, 9] ❶
suma = 0
x = 0
while x < 5: ❷
    suma += notas[x] ❸
    x += 1 ❹
print("Media: %5.2f" % (suma/x))
```

Creamos la lista de notas en ❶. En ❷ creamos la estructura de repetición para variar el valor de x y continuar mientras este sea menor que 5. Recuerde que una lista de cinco elementos contiene índices de 0 a 4. Por eso inicializamos x=0 en la línea anterior. En ❸ adicionamos el valor de notas[0] a la suma y después notas[1], notas[2], notas[3] y notas[4]; un elemento en cada repetición. Para eso utilizamos el valor de x como índice, y lo incrementamos de 1 en ❹. Una gran ventaja de este programa es que no necesitamos declarar cinco variables para guardar las cinco notas. Todas las notas fueron almacenadas en la lista, utilizando un índice para identificar o acceder a cada valor.

Veamos una modificación de ese ejemplo, pero esta vez vamos a leer las notas una por una. El programa modificado es presentado en la lista 6.6.

► Lista 6.6 – Cálculo de la media con notas digitadas

```
notas = [0, 0, 0, 0, 0] ❶
suma = 0
```



```

x = 0
while x < 5:
    notas[x] = float(input("Nota %d:" % x)) ❷
    suma += notas[x]
    x += 1
x = 0 ❸
while x < 5: ❹
    print("Nota %d: %6.2f" % (x, notas[x]))
    x += 1
print("Media: %5.2f" % (suma / x))

```

En ❶ creamos la lista de notas con cinco elementos, todos cero. En ❷ utilizamos la repetición para leer las notas del alumno y almacenarlas en la lista de notas. Vea que adicionamos `notas[x]` a la suma ya en la línea siguiente. Terminada la primera repetición habremos llenado la lista de notas. Para imprimir la lista de notas, reinicializamos el valor de la variable `x` en 0 ❸ y creamos otra estructura de repetición ❹.

Ejercicio 6.1 Modifique el programa de la lista 6.6 para leer 7 notas en vez de 5.

6.1 Trabajando con índices

Veamos otro ejemplo: un programa que lee cinco números, los almacena en una lista y después solicita que el usuario elija un número para mostrar. El objetivo es, por ejemplo, leer 15, 12, 5, 7 y 9 y almacenarlos en la lista. Si el usuario digita 2, se imprimirá el segundo número digitado; si digita 3, el tercero; y así sucesivamente. Observe que el índice del primer número es 0 y no 1: esa pequeña conversión será hecha en el programa de la lista 6.7.

► Lista 6.7 – Presentación de números

```

números=[0, 0, 0, 0, 0]
x = 0
while x < 5:
    números[x] = int(input("Número %d:" % (x + 1))) ❶
    x += 1
while True:
    elegido = int(input("Qué posición quiere imprimir usted (0 para salir): "))
    if elegido == 0:
        break
    print("usted eligió el número: %d" % (números[elegido - 1])) ❷

```

Ejecute el programa de la lista 6.7 y pruebe algunos valores. Observe que en ❶ adicionamos 1 a `x` para poder imprimir ‘Número 1...5’ y no hacerlo a partir de 0. Eso es importante porque empezar a contar desde 0 no es natural para la mayoría de las personas. Vea que aún imprimiendo `x + 1` para el

usuario, la atribución está hecha para `números[x]` porque nuestras listas empiezan en 0. En ❷, hicimos la operación inversa. Cuando el usuario elige el número a imprimir, él hace una elección entre 1 y 5. Como 1 es el elemento 0; 2 es el elemento 1; y así sucesivamente, disminuimos el valor de la elección en 1 para obtener el índice de notas.

6.2 Copia y rebanado de listas

Aunque las listas en Python sean un recurso muy poderoso, todo poder trae responsabilidades. Uno de los efectos colaterales de las listas aparece cuando tratamos de hacer copias. Veamos un test en el intérprete, en la lista 6.8.

► Lista 6.8 – Intento de copiar listas

```
>>> L=[1, 2, 3, 4, 5]
>>> V=L
>>> L
[1, 2, 3, 4, 5]
>>> V
[1, 2, 3, 4, 5]
>>> V[0]=6
>>> V
[6, 2, 3, 4, 5]
>>> L
[6, 2, 3, 4, 5]
```

Vea que al modificar `V`, modificamos también el contenido de `L`. Eso sucede porque una lista en Python es un objeto; y cuando atribuimos un objeto a otro solo estamos copiando la misma referencia de la lista, y no sus datos en sí. En este caso `V` funciona como un apodo de `L`; o sea que `V` y `L` son la misma lista. Veamos qué sucede en el gráfico de la figura 6.1.

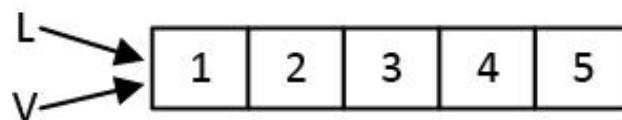


Figura 6.1 – Dos variables referenciando la misma lista.

Cuando modificamos `V[0]`, estamos modificando el mismo valor de `L[0]`, pues ambos son referencias, o apodos para la misma lista guardada en la memoria.

Dependiendo de la aplicación, ese efecto puede ser deseado o no. Para crear una copia independiente de una lista, utilizaremos otra sintaxis. Veamos el resultado de las operaciones de la lista 6.9.

► Lista 6.9 – Copia de listas

```
>>> L=[1, 2, 3, 4, 5]
>>> V=L[:]
>>> V[0]=6
```

```
>>> L
[1, 2, 3, 4, 5]
>>> V
[6, 2, 3, 4, 5]
```

Al escribir `L[:]`, nos estamos refiriendo a una nueva copia de `L`. Así `L` y `V` se refieren a áreas diferentes en la memoria, lo cual nos permite alterarlas de forma independiente, como en la figura 6.2.

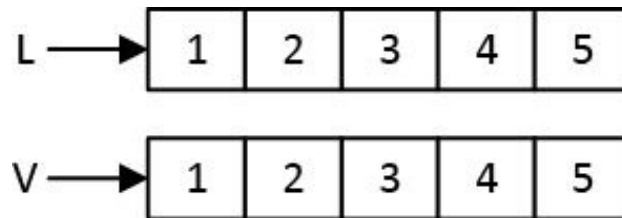


Figura 6.2 – Dos variables referenciando dos listas.

Podemos también rebanar una lista, del mismo modo que hicimos con cadenas de caracteres en el capítulo 3. Veamos algunos ejemplos del intérprete en la lista 6.10.

► Lista 6.10 – Rebanado de listas

```
>>> L=[1, 2, 3, 4, 5]
>>> L[0:5]
[1, 2, 3, 4, 5]
>>> L[:5]
[1, 2, 3, 4, 5]
>>> L[:-1]
[1, 2, 3, 4]
>>> L[1:3]
[2, 3]
>>> L[1:4]
[2, 3, 4]
>>> L[3:]
[4, 5]
>>> L[:3]
[1, 2, 3]
>>> L[-1]
5
>>> L[-2]
4
```

Vea que índices negativos también funcionan. Un índice negativo comienza a contar a partir del último elemento, pero observe que empezamos de `-1`. Así `L[0]` representa el primer elemento; `L[-1]`, el último; `L[-2]`, el penúltimo, y así sucesivamente.

6.3 Tamaño de listas

Podemos usar la función `len` con listas. El valor retornado es igual al número de elementos de la lista. Vea algunas pruebas en la lista 6.11.

► Lista 6.11 – Tamaño de listas

```
>>> L=[12, 9, 5]
>>> len(L)
3
>>> V=[]
>>> len(V)
0
```

La función `len` puede ser utilizada en repeticiones para controlar el límite de los índices. Vea el ejemplo en la lista 6.12.

► Lista 6.12 – Repetición con tamaño fijo de la lista

```
L=[1, 2, 3]
x = 0
while x < 3:
    print(L[x])
    x += 1
```

Eso puede ser reescrito como en la lista 6.13.

► Lista 6.13 – Repetición con tamaño de la lista usando `len`

```
L=[1, 2, 3]
x = 0
while x < len(L):
    print(L[x])
    x += 1
```

La ventaja es que si cambiamos `L` a:

```
L=[7, 8, 9, 10, 11, 12]
```

el resto del programa continuaría funcionando, pues utilizamos la función `len` para calcular el tamaño de la lista. Observe que el valor retornado por la función `len` es un número que no puede ser utilizado como índice, pero que es perfecto para probar los límites de una lista, como hicimos en la lista 6.13. Eso sucede porque `len` retorna la cantidad de elementos en la lista y nuestros índices empiezan a ser numerados desde 0 (cero). Así, los índices válidos de una lista (`L`) varían de 0 hasta el valor de `len(L) - 1`.

6.4 Adición de elementos

Una de las principales ventajas de trabajar con listas es poder adicionar nuevos elementos durante la

ejecución del programa. Veamos un test en el intérprete (Lista 6.14).

Para adicionar un elemento al fin de la lista utilizaremos el método **append**. En Python, llamamos un método escribiendo su nombre después del nombre del objeto. Como las listas son objetos, siendo **L** la lista, tendremos **L.append(valor)**. Los métodos son recursos de orientación a objetos, soportados y muy usados en Python. Puede imaginar un método como una función del objeto. Cuando es invocado, él ya sabe a qué objeto nos estamos refiriendo, pues lo informamos a la izquierda del punto. Hablaremos más sobre métodos en el capítulo 10; por ahora, observe cómo los utilizamos y sepa que son diferentes a las funciones.

► Lista 6.14 – Adición de elementos a la lista

```
>>> L=[]
>>> L.append("a")
>>> L
['a']
>>> L.append("b")
>>> L
['a', 'b']
>>> L.append("c")
>>> L
['a', 'b', 'c']
>>> len(L)
3
```

Veamos un programa que lee números hasta que 0 sea digitado. Ese programa después los imprimirá en el mismo orden en que fueron digitados (Lista 6.15).

► Lista 6.15 – Adición de elementos a la lista

```
L = []
while True:
    n = int(input("Digite un número (0 sale):"))
    if n == 0:
        break
    L.append(n)
x = 0
while x < len(L):
    print(L[x])
    x += 1
```

Este simple programa es capaz de leer e imprimir un número inicialmente indeterminado de valores. Esto es posible porque adicionamos elementos a la lista **L**, según fue necesario.

Otra forma de adicionar elementos a una lista es con “adición de listas” (Lista 6.16).

Cuando adicionamos solo un elemento, tanto `L.append(1)` como `L+[1]` producen el mismo resultado. Observe que en `L+[1]` escribimos el elemento a adicionar dentro de una lista (`[1]`), y que, en `append`, solo 1. Eso porque, cuando adicionamos una lista a otra, el intérprete ejecuta un método llamado `extend` que adiciona los elementos de una lista a otra. Veamos algunos ejemplos en la lista 6.17.

► Lista 6.16 – Adición de listas

```
>>> L=[]
>>> L=L+[1]
>>> L
[1]
>>> L+= [2]
>>> L
[1, 2]
>>> L+= [3,4,5]
>>> L
[1, 2, 3, 4, 5]
```

► Lista 6.17 – Adición de elementos y listas

```
>>> L=["a"]
>>> L.append("b")
>>> L
['a', 'b']
>>> L.extend(["c"])
>>> L
['a', 'b', 'c']
>>> L.append(["d","e"])
>>> L
['a', 'b', 'c', ['d', 'e']]
>>> L.extend(["f","g","h"])
>>> L
['a', 'b', 'c', ['d', 'e'], 'f', 'g', 'h']
```

El método `extend` no acepta parámetros que no sean listas. Si usted utiliza el método `append` con una lista como parámetro, en vez de adicionar los elementos al final de la lista, `append` adicionará la lista entera, pero como un nuevo elemento único. Tendremos entonces listas dentro de listas (Lista 6.18).

► Lista 6.18 – Adición de elementos y listas con `append`

```
>>> L=["a"]
>>> L.append(["b"])
```

```
>>> L.append(["c","d"])
>>> len(L)
3
>>> L[1]
['b']
>>> L[2]
['c', 'd']
>>> len(L[2])
2
>>> L[2][1]
'd'
```

Este concepto es interesante, pues permite la utilización de estructuras de datos más complejas, como matrices, árboles y registros. Por ahora vamos a utilizar este recurso para almacenar múltiples valores por elemento.

Ejercicio 6.2 Haga un programa que lea dos listas y que genere una tercera con los elementos de las dos primeras.

Ejercicio 6.3 Haga un programa que recorra dos listas y genere una tercera sin elementos repetidos.

6.5 Remoción de elementos de la lista

Así como el tamaño de la lista puede variar, permitiendo la adición de nuevos elementos, también podemos retirar algunos elementos de la lista, e incluso retirar todos los elementos. Para eso utilizaremos la instrucción **del** (Lista 6.19).

► Lista 6.19 – Remoción de elementos

```
>>> L=["a","b","c"]
>>> del L[1]
>>> L
['a', 'c']
>>> del L[0]
>>> L
['c']
```

Es importante notar que el elemento excluido no ocupa más un lugar en la lista, haciendo que los índices sean reorganizados; o mejor dicho, que pasen a ser calculados sin ese elemento.

Podemos también borrar rebanadas enteras de una sola vez (Lista 6.20).

► Lista 6.20 – Remoción de rebanadas

```
>>> L=list(range(101))
```

```
>>> del L[1:99]
>>> L
[0, 99, 100]
```

6.6 Usando listas como colas

Una lista puede ser utilizada como cola si obedecemos ciertas reglas de inclusión y eliminación de elementos. En una cola, la inclusión siempre es realizada al final de la lista, y las remociones son hechas al comienzo. Decimos que el primero en llegar es el primero en salir (*FIFO – First In First Out*).

Es más simple de entender si imaginamos una cola de banco. Cuando la agencia abre por la mañana, la cola está vacía. Cuando los clientes empiezan a llegar, van directamente al final de la cola. Las cajas entonces empiezan a atender a esos clientes por orden de llegada; o sea, el cliente que llegó primero será atendido primero. Una vez que el cliente es atendido, sale de la cola. Entonces un nuevo cliente pasa a ser el primero de la cola y el próximo en ser atendido.

Para escribir algo similar en Python, imaginaremos una lista de clientes representando la cola, donde el valor de cada elemento es igual al orden de llegada del cliente. Vamos a imaginar una lista inicial con 10 clientes. Si otro cliente llega, realizaremos un **append** para que él sea introducido al final de la cola (**cola.append(último)**). Para retirar un cliente de la cola y atenderlo, podríamos hacer **del cola[0]**, sin embargo eso borraría al cliente de la cola. Si queremos retirarlo de la cola y, al mismo tiempo, obtener el elemento retirado, podemos utilizar el método **pop** **cola.pop(0)**. El método **pop** devuelve el valor del elemento y lo excluye de la cola. Pasamos 0 como parámetro para indicar que queremos excluir el primer elemento. Vea el programa completo en la lista 6.21.

► Lista 6.21 – Simulación de una cola de banco

```
último = 10
cola = list(range(1, último + 1))
while True:
    print("\nExisten %d clientes en la cola" % len(cola))
    print("Cola actual:", cola)
    print("Digite F para adicionar un cliente al final de la cola,")
    print("o A para realizar la atención. S para salir.")
    operación = input("Operación (F, A o S):")
    if operación == "A":
        if(len(cola))>0:
            atendido = cola.pop(0)
            print("Cliente %d atendido" % atendido)
        else:
            print(";Cola vacía! Nadie para atender.")
    elif operación == "F":
        último+=1 # Incrementa el ticket del nuevo cliente
```



```

cola.append(último)
elif operación == "S":
    break
else:
    print("¡Operación inválida! ¡Dígite solo F, A o S!")

```

Ejercicio 6.4 ¿Qué sucede cuando no verificamos si la lista está vacía antes de llamar el método **pop**?

Ejercicio 6.5 Altere el programa de la lista 6.21 de modo de poder trabajar con varios comandos digitados de una sola vez. Actualmente, solo un comando puede ser introducido por vez. Altérelolo de modo de considerar la operación como una cadena de caracteres.

Ejemplo: FFFAAAS significaría tres llegadas de nuevos clientes, tres atenciones y, finalmente, la salida del programa.

Ejercicio 6.6 Modifique el programa para trabajar con dos colas. Para facilitar su trabajo, considere el comando A para atención de la cola 1; y B, para atención de la cola 2. El mismo para la llegada de clientes: F para cola 1; y G, para cola 2.

6.7 Uso de listas como pilas

Una pila tiene una política de acceso bien definida: nuevos elementos son adicionados al tope. El retiro de elementos también es hecho desde el tope.

Imagine una pila de platos para lavar. Retiramos el plato que está en el tope de la pila para lavar, y si llegan algunos platos más, serán también adicionados o apilados en el tope; uno sobre otro. En la pila, el último elemento en llegar es el primero a salir (*LIFO -Last In First Out*). Por ahora, vamos a concentrarnos en las pilas de platos. Vea el programa de la lista 6.22, que simula una pileta de cocina llena de platos.

Usted debe haber notado que el ejemplo de la lista 6.22 es muy parecido al programa de la lista 6.21. Esto es así porque la gran diferencia entre pilas y colas es el elemento que elegimos para retirar. En una cola, el primer elemento es retirado primero. En las pilas, se retira a partir del último elemento. El único cambio en Python es el valor que pasamos por el método **pop**. En el caso de una pila, como retiramos el último elemento, pasamos **-1** a **pop**.

► Lista 6.22 – Pila de platos

```

plato = 5
pila = list(range(1,plato+1))
while True:
    print("\nExisten %d platos en la pila" % len(pila))
    print("Pila actual:", pila)
    print("Dígite A para apilar un nuevo plato,")
    print("o R para retirar de la pila. S para salir.")

```

```

operación = input("Operación (A, R o S):")
if operación == "R":
    if(len(pila))>0:
        lavado = pila.pop(-1)
        print("Plato %d lavado" % lavado)
    else:
        print(";Pila vacía! Nada para lavar.")
elif operación == "A":
    plato += 1 # Nuevo plato
    pila.append(plato)
elif operación == "S":
    break
else:
    print(";Operación inválida! ;Digite solo A, R o S!")

```

Puede, también, observar el uso de pilas con su browser de internet. Abra algunas páginas haciendo clic en sus links. Observe cómo el historial de navegación se modifica. Cada nuevo link adiciona una página a su historial de navegación. Si hace clic en volver una página, el browser utilizará la última página que ingresó en el historial. En ese caso, el historial del browser funciona como una pila. En realidad es un poco más complejo, pues permite que volvamos varias veces y que después podamos volver en el otro sentido si es necesario (avanzar). Ahora haga otro test: vuelva dos o tres veces y después visite un nuevo link. Esta vez, el browser debe haber desactivado la función de avance, ya que el historial cambió (usted cambió de dirección). Trate de entender ese proceso como dos pilas; una a la izquierda y otra a la derecha. Cuando usted elige volver, sacamos un elemento de la pila a la izquierda. Al visitar un nuevo link, adicionamos un nuevo elemento a esa misma pila. Para simular la opción de avanzar, imagine que al retirar un elemento de la pila a la izquierda, debemos adicionarlo a la de la derecha. Si escogemos una dirección nueva, borramos la pila de la derecha.

Esas mismas operaciones pueden ser simuladas con otras estructuras, tales como una lista simple y una variable conteniendo nuestra posición en el historial de navegación. A cada nueva dirección, adicionaríamos un nuevo elemento a la lista y actualizaríamos nuestra posición. Al volver, disminuiríamos la posición y la incrementaríamos en el caso de avance. Si cambiásemos de dirección, bastaría borrar los elementos entre la posición actual y el fin de la lista antes de adicionar el nuevo elemento.

La importancia de aprender a manipular listas como colas o pilas es entender algoritmos más complejos en el futuro.

Ejercicio 6.7 Haga un programa que lea una expresión con paréntesis. Usando pilas, verifique si los paréntesis fueron abiertos y cerrados en el orden correcto. Ejemplo:

(())	OK
()()()())	OK
())	Error

Puede adicionar elementos a la pila siempre que encuentre abre paréntesis y retirar en cada cierra paréntesis. Al retirar, verifique si el tope de la pila es un abre paréntesis. Si la expresión es correcta, su pila estará vacía al final.

6.8 Investigación

Podemos investigar si un elemento está o no en una lista, verificando si el valor buscado está presente desde el primero al último elemento. Veamos la lista 6.23.

► Lista 6.23 – Investigación secuencial

```
L = [15, 7, 27, 39]
p = int(input("Digite el valor a buscar:"))
encontró = False ❶
x = 0
while x < len(L):
    if L[x] == p:
        encontró=True ❷
        break ❸
    x += 1
if encontró: ❹
    print("%d encontrado en la posición %d" % (p, x))
else:
    print("%d no encontrado" % p)
```

La investigación simplemente compara todos los elementos de la lista con el valor buscado, interrumpiendo la repetición al encontrar el primer elemento cuyo valor es igual al buscado. Es importante saber que podemos encontrar o no lo que buscamos, por eso la utilización de la variable **encontró** en ❶. Esa variable del tipo lógico (booleano) será utilizada para verificar si salimos de la repetición por haber encontrado lo que buscábamos o simplemente porque visitamos todos los elementos sin encontrar el valor buscado. Vea que **encontró** está marcada como **True** en ❷, pero dentro de **if** con la condición de investigación, y antes del **break** en ❸. De esa forma, **encontró** solo será **True** si algún elemento es igual al valor buscado. En ❹, verificamos el valor de **encontró** para decidir lo que vamos imprimir.

Ejercicio 6.8 Modifique el primer ejemplo (Lista 6.23) de modo de realizar la misma tarea, pero sin utilizar la variable **encontró**. Consejo: observe la condición de salida del **while**.

Ejercicio 6.9 Modifique el ejemplo para investigar dos valores. En vez de solo **p**, lea otro valor **v** que también será buscado. En la impresión, indique cuál de los dos valores fue encontrado primero.

Ejercicio 6.10 Modifique el programa del ejercicio 6.9 de modo de investigar **p** y **v** en toda la lista e informando al usuario las posiciones donde **p** y **v** fueron encontrados.

6.9 Usando for

Python presenta una estructura de repetición especialmente proyectada para recorrer listas. La instrucción **for** funciona de forma parecida a **while**, pero en cada repetición utiliza un elemento diferente de la lista.

En cada repetición el próximo elemento de la lista es utilizado, lo cual se repite hasta el fin de la lista. Vamos a escribir un programa que utilice **for** para imprimir todos los elementos de una lista (Lista 6.24).

► Lista 6.24 – Impresión de todos los elementos de la lista con for

```
L = [8, 9, 15]
for e in L: ❶
    print(e) ❷
```

Cuando empezamos a ejecutar el **for** en ❶, tenemos **e** igual al primer elemento de la lista, en este caso 8, o sea, **L[0]**. En ❷ imprimimos 8, y la ejecución del programa vuelve a ❶, donde **e** pasa a valer 9, o sea, **L[1]**. En la próxima repetición **e** valdrá 15, o sea, **L[2]**. Después de imprimir el último número la repetición concluye, pues no tenemos más elementos para sustituir. Si tuviésemos que hacer la misma tarea con **while**, tendríamos que escribir un programa como el de la lista 6.25.

► Lista 6.25 – Impresión de todos los elementos de la lista con while

```
L = [8, 9, 15]
x = 0
while x < len(L):
    e = L[x]
    print(e)
    x += 1
```

Aunque la instrucción **for** facilite nuestro trabajo, no sustituye completamente a **while**. Dependiendo del problema, utilizaremos **for** o **while**. Normalmente utilizaremos **for** cuando queramos procesar los elementos de una lista, uno por uno. Por su parte, **while** está indicado para repeticiones en las cuales no sabemos aún cuántas veces vamos a repetir o dónde manipularemos los índices de forma no secuencial.

Vale recordar que la instrucción **break** también interrumpe el **for**. En la lista 6.26, vemos la investigación, usando **for**.

► Lista 6.26 – Investigación usando for

```
L = [7, 9, 10, 12]
p = int(input("Digite un número a investigar:"))
for e in L:
    if e == p:
        print("¡Elemento encontrado!")
        break ❶
```

else: ❷

```
print("Elemento no encontrado.")
```

Utilizamos la instrucción **break** para interrumpir la búsqueda después de encontrar el primer elemento en ❶. En ❷ utilizamos un **else**, parecido al de la instrucción **if**, para imprimir el mensaje informando que el elemento no fue encontrado. El **else** debe ser escrito en la misma columna de **for** y solo será ejecutado si todos los elementos de la lista son visitados, o sea, si no se utiliza la instrucción **break**, dejando que **for** termine normalmente.

Ejercicio 6.11 Modifique el programa de la lista 6.15 usando **for**. Explique porqué no todos los **while** pueden ser transformados en **for**.

6.10 Range

Podemos utilizar la función **range** para generar listas simples. La función **range** no retorna una lista propiamente dicha, sino un generador o *generator*. Por ahora basta entender cómo podemos usarla. Imagine un programa simple que imprime de 0 a 9 en la pantalla (Lista 6.27).

► Lista 6.27 – Uso de la función range

```
for v in range(10):  
    print(v)
```

Ejecute el programa y vea el resultado. La función **range** generó números de 0 a 9 porque pasamos 10 como parámetro. Ella normalmente genera valores a partir de 0, luego, al especificar 10, estamos solo informando dónde parar. Pruebe cambiar de 10 a 20. El programa debe imprimir de 0 a 19. Ahora pruebe 20000. La ventaja de utilizar la función **range** es la de generar listas eficientemente, tal como se muestra en el ejemplo, sin necesidad de escribir los 20.000 valores en el programa.

Con la misma función **range**, también podemos indicar cuál es el primer número a generar. Para eso utilizaremos dos parámetros: comienzo y fin (Lista 6.28).

► Lista 6.28 – Uso de la función range con intervalos

```
for v in range(5, 8):  
    print(v)
```

Usando 5 como comienzo y 8 como fin, vamos a imprimir los números 5, 6 y 7. La notación aquí para el fin es la misma utilizada con rebanadas, o sea, el fin es un intervalo abierto no incluido en la franja de valores.

Si agregamos un tercer parámetro a la función **range**, podremos saltar entre los valores generados; por ejemplo, **range(0, 10, 2)** genera los pares entre 0 y 10, pues comienza de 0 y adiciona 2 a cada elemento. Veamos un ejemplo donde generamos los 10 primeros múltiplos de 3 (Lista 6.29).

► Lista 6.29 – Uso de la función range con saltos

```
for t in range(3, 33, 3):  
    print(t, end=" ")
```

```
print()
```

Observe que un generador como el retornado por la función `range` no es exactamente una lista. Aunque sea usado de forma parecida, en realidad es un objeto de otro tipo. Para transformar un generador en lista, utilice la función `list` (Lista 6.30).

► Lista 6.30 – Transformación del resultado de `range` en una lista

```
L = list(range(100, 1100, 50))  
print(L)
```

Volviendo a la lista 6.29, observe que utilizamos una construcción especial con la función `print`, donde `t` es el valor que queremos imprimir, pero con `end=" "`, que le indica a la función no saltar de línea después de la impresión. `end` es, en realidad, un parámetro opcional de la función `print`. Veremos más sobre eso cuando estudiemos funciones en el capítulo 8. Vea también que, para saltar la línea en el final del programa, hicimos una llamada a `print()` sin ningún parámetro.

6.11 Enumerate

Con la función `enumerate` podemos ampliar las funcionalidades de `for` fácilmente. Veamos cómo imprimir una lista, donde tendremos el índice entre corchetes y el valor a su derecha (Lista 6.31).

► Lista 6.31 – Impresión de índices sin usar la función `enumerate`

```
L = [5, 9, 13]  
x = 0  
for e in L:  
    print("[%d] %d" % (x,e))  
    x += 1
```

Vea el mismo programa, pero utilizando la función `enumerate` en la lista (Lista 6.32).

► Lista 6.32 – Impresión de índices usando la función `enumerate`

```
L = [5, 9, 13]  
for x, e in enumerate(L):  
    print("[%d] %d" % (x, e))
```

La función `enumerate` genera una tupla en la cual el primer valor es el índice y el segundo es el elemento de la lista enumerada. Al utilizar `x, e` en `for`, indicamos que el primer valor de la tupla debe ser colocado en `x`, y el segundo en `e`. Así, en la primera iteración tendremos la tupla (0,5), donde `x = 0` y `e = 5`. Esto es posible porque Python permite el desempaquetamiento de valores de una tupla, atribuyendo un elemento de la tupla a cada variable en `for`. Por tanto tenemos que cada iteración de `for` es equivalente a `x, e = (0, 5)`, en que el generador `enumerate` retorna cada vez una nueva tupla. Los próximos valores retornados son (1, 9) y (2, 13), respectivamente. Pruebe sustituir `x, e` en la lista 6.30 por `z`. Antes de `print`, haga `x, e = z`. Adicione un `print` más para exhibir también el valor de `z`.

6.12 Operaciones con listas

Podemos recorrer una lista de modo de verificar el menor y el mayor valor (Lista 6.33).

► Lista 6.33 – Verificación del mayor valor

```
L = [1, 7, 2, 4]
máximo = L[0] ❶
for e in L:
    if e > máximo:
        máximo = e
print(máximo)
```

En ❶, utilizamos un pequeño truco, inicializando el máximo con el valor del primer elemento. Necesitamos un valor para máximo antes de utilizarlo en a comparación con `if`. Si usásemos 0, no tendríamos problema, siempre que nuestra lista no tenga valores negativos.

Ejercicio 6.12 Altere el programa de la lista 6.33 de modo de imprimir el menor elemento de la lista.

Ejercicio 6.13 La lista de temperaturas de Mons, en Bélgica, fue almacenada en la lista `T = [-10, -8, 0, 1, 2, 5, -2, -4]`. Haga un programa que imprima la menor y la mayor temperatura, así como la temperatura media.

6.13 Aplicaciones

Veamos una situación en la cual tenemos que seleccionar los elementos de una lista de modo de copiarlos a otras dos listas. Para simplificar el problema, imagine que los valores estén inicialmente en la lista `V`, pero que deban ser copiados para la `P`, si son pares; o para la `I`, si son impares. Vea el programa que resuelve ese problema, en la lista 6.34.

► Lista 6.34 – Copia de elementos a otras listas

```
V = [9, 8, 7, 12, 0, 13, 21]
P = []
I = []
for e in V:
    if e % 2 == 0:
        P.append(e)
    else:
        I.append(e)
print("Pares: ", P)
print("Impares: ", I)
```

Veamos ahora un programa que controla la utilización de las salas de un cine. Imagine que la lista `Salas = [10, 2, 1, 3, 0]` contenga el número de lugares vacíos en las salas 1, 2, 3, 4 y 5, respectivamente. Ese programa leerá el número de la sala y la cantidad de lugares solicitados. El

programa debe informar si es posible vender el número de lugares solicitados; dicho de otro modo, si aún hay lugares libres. En caso que sea posible vender los billetes, actualizará el número de lugares libres. La salida ocurre cuando se digita 0 en el número de la sala (Lista 6.35).

► Lista 6.35 – Control de la utilización de salas de un cine

```
lugares_vacios=[10, 2, 1, 3, 0]
while True:
    sala = int(input("Sala (0 sai): "))
    if sala == 0:
        print("Fin")
        break
    if sala > len(lugares_vacios) or sala < 1:
        print("Sala inválida")
    elif lugares_vacios[sala - 1] == 0:
        print(";Disculpe, sala llena!")
    else:
        lugares = int(input("Cuántos lugares desea usted (%d vacíos):"
            % lugares_vacios[sala - 1]))
        if lugares > lugares_vacios[sala-1]:
            print("Ese número de lugares no está disponible.")
        elif lugares < 0:
            print("Número inválido")
        else:
            lugares_vacios[sala - 1] -= lugares
            print("%d lugares vendidos" % lugares)
print("Utilización de las salas")
for x, l in enumerate(lugares_vacios):
    print("Sala %d - %d lugar(es) vacío(s)" % (x + 1, l))
```

6.14 Listas con cadenas de caracteres

En el capítulo 3 vimos que las cadenas de caracteres pueden ser indexadas letra por letra. Las listas en Python funcionan del mismo modo, permitiendo el acceso a varios valores y volviéndose una de las principales estructuras de programación del lenguaje.

Veamos ahora otro ejemplo de listas, pero utilizando cadenas de caracteres, en la lista 6.36. En ese caso, *S* es una lista, y cada elemento, una cadena de caracteres.

► Lista 6.36 – Listas con cadenas de caracteres

```
>>> S = ["manzanas", "peras", "kiwis"]
>>> print(len(S))
```



```
>>> print(S[0])
manzanas
>>> print(S[1])
peras
>>> print(S[2])
kiwis
```

El programa de la lista 6.37 lee e imprime una lista de compras hasta que sea digitado fin.

► Lista 6.37 – Leyendo e imprimiendo una lista de compras

```
compras = []
while True:
    product = input("Producto:")
    if producto == "fin":
        break
    compras.append(producto)
for p in compras:
    print(p)
```

6.15 Listas dentro de listas

Un factor interesante es que podemos acceder a las cadenas de caracteres dentro de la lista, letra por letra, usando un segundo índice. Veamos las listas 6.38 y 6.39.

► Lista 6.38 – Listas con cadenas de caracteres, accediendo a letras

```
>>> S=["manzanas", "peras", "kiwis"]
>>> print(S[0][0])
m
>>> print(S[0][1])
a
>>> print(S[1][1])
e
>>> print(S[2][2])
w
```

► Lista 6.39 – Impresión de una lista de cadenas de caracteres, letra por letra

```
L=["manzanas", "peras", "kiwis"]
for s in L:
    for letra in s:
        print(letra)
```

Eso nos lleva a otra ventaja de las listas en Python: listas dentro de listas. Tenemos también que los

elementos de una lista no necesitan ser del mismo tipo. Veamos un ejemplo donde tendríamos una lista de compras en la lista 6.40. El primer elemento sería el nombre del producto; el segundo, la cantidad; y el tercero, su precio.

► Lista 6.40 – Listas con elementos de tipos diferentes

```
producto1 = ["manzana", 10, 0.30]
producto2 = ["pera", 5, 0.75]
producto3 = ["kiwi", 4, 0.98]
```

Así, `producto1`, `producto2`, `producto3` serían tres listas con tres elementos cada una. ¡Observe que el primer elemento es del tipo cadena de caracteres; el segundo del tipo entero; y el tercero del tipo punto flotante (*float*)!

Veamos ahora otra lista, en la lista 6.41.

► Lista 6.41 – Listas de listas

```
producto1 = ["manzana", 10, 0.30]
producto2 = ["pera", 5, 0.75]
producto3 = ["kiwi", 4, 0.98]
compras = [producto1, producto2, producto3]
print(compras)
```

Ahora tenemos una lista llamada `compras`, también con tres elementos, pero cada elemento es una lista aparte. Para imprimir esa lista, tendríamos el programa de la lista 6.42.

► Lista 6.42 – Impresión de las compras

```
producto1 = ["manzana", 10, 0.30]
producto2 = ["pera", 5, 0.75]
producto3 = ["kiwi", 4, 0.98]
compras = [producto1, producto2, producto3]
for e in compras:
    print("Producto: %s" % e[0])
    print("Cantidad: %d" % e[1])
    print("Precio: %5.2f" % e[2])
```

Del mismo modo podríamos haber accedido al precio del segundo elemento con `compras[1][2]`. Veamos ahora un programa completo, capaz de preguntar nombre del producto, cantidad y precio y, al final, imprimir una lista de compras completa.

► Lista 6.43 – Creación e impresión de la lista de compras

```
compras = []
while True:
    producto = input("Producto: ")
    if producto == "fin":
```

break

```
cantidad = int(input("Cantidad: "))  
precio = float(input("Precio: "))  
compras.append([producto, cantidad, precio])
```

suma = 0.0

for e in compras:

```
    print("%20s x%5d %5.2f %6.2f" % (e[0],  
                                     e[1],e[2],  
                                     e[1] * e[2]))
```

```
    suma += e[1] * e[2]
```

```
print("Total: %7.2f" % suma)
```

6.16 Ordenamiento

Hasta ahora, los elementos de nuestras listas presentan el mismo orden en que fueron digitados, sin ningún ordenamiento. Para ordenar una lista, realizaremos una operación semejante a la de la investigación, pero cambiando el orden de los elementos cuando sea necesario. Un algoritmo muy simple de ordenamiento es el “*Bubble Sort*”, o método de burbuja, fácil de entender y aprender. Por ser lento, no debe utilizarlo con listas grandes.

El ordenamiento por el método de burbuja consiste en comparar dos elementos cada vez. Si el valor del primer elemento es mayor que el del segundo, cambiarán de posición. Esa operación se repite con el siguiente elemento hasta el final de la lista. El método de burbuja exige que recorramos la lista varias veces. Por eso utilizaremos un marcador para saber si llegamos al final de la lista, cambiando o no algún elemento. Ese método tiene otra propiedad, que es posicionar el mayor elemento en la última posición de la lista, o sea, en su posición correcta. Eso permite eliminar un elemento del final de la lista en cada pasaje completo. Veamos el programa de la lista 6.44.

► Lista 6.44 – Ordenamiento por el método de burbuja

```
L = [7, 4, 3, 12, 8]
```

```
fin = 5 ❶
```

```
while fin > 1: ❷
```

```
    cambió = False ❸
```

```
    x = 0 ❹
```

```
    while x < (fin - 1): ❺
```

```
        if L[x] > L[x + 1]: ❻
```

```
            cambió = True ❼
```

```
            temp = L[x] ❸
```

```
            L[x] = L[x + 1]
```

```
            L[x+1] = temp
```

```
        x += 1
```

```
    if not cambió: ❾
```

```

break
fin -= 1 ❶
for e in L:
    print(e)

```

Ejecute el programa y trate de entender cómo funciona. En ❶, utilizamos la variable **fin** para marcar la cantidad de elementos de la lista. Necesitamos marcar el fin o la posición del último elemento porque en el “*Bubble Sort*” no necesitamos verificar el último elemento después de un pasaje completo. En ❷, verificamos si **fin > 1**, pues como comparamos el elemento actual (**L[x]**) con el siguiente (**L[x + 1]**), necesitamos, como mínimo, dos elementos. Utilizamos la variable **cambió** en ❸ para indicar que no realizamos ningún cambio. El valor de **cambió** será utilizado más tarde para verificar si ya tenemos la lista ordenada o no. La variable **x** ❹ será utilizada como índice, comenzando en la posición 0. Nuestra condición del segundo **while** ❺ es especial, pues tenemos que garantizar un próximo elemento para compararlo con el actual. Esto hace que la condición de salida de ese **while** sea **x < (fin - 1)**, o mejor dicho, que **x** sea anterior al último elemento. En ❻ tenemos la comparación en sí, donde **x** es nuestra posición actual, y el próximo elemento es el de índice **x + 1**. Como estamos ordenando en orden creciente, deseamos que el próximo elemento siempre sea mayor que el actual, garantizando de esa forma el ordenamiento de la lista. Como comparamos solo dos elementos por vez, tenemos que revisar la lista varias veces, hasta que todos los elementos estén en sus posiciones. Si la condición de ❻ es verdadera, esos elementos están fuera de orden. En ese caso, debemos invertir o cambiar la posición de los dos elementos. En ❼ marcamos que efectuamos un cambio y, en ❽, utilizamos una variable auxiliar o temporal para cambiar los dos valores de posición.

El cambio de valores entre dos variables se muestra en las figuras 6.3, 6.4 y 6.5. Como cada variable solo guarda un valor por vez, utilizaremos una tercera variable temporal (**temp**) para almacenar el valor de una de ellas durante el cambio.

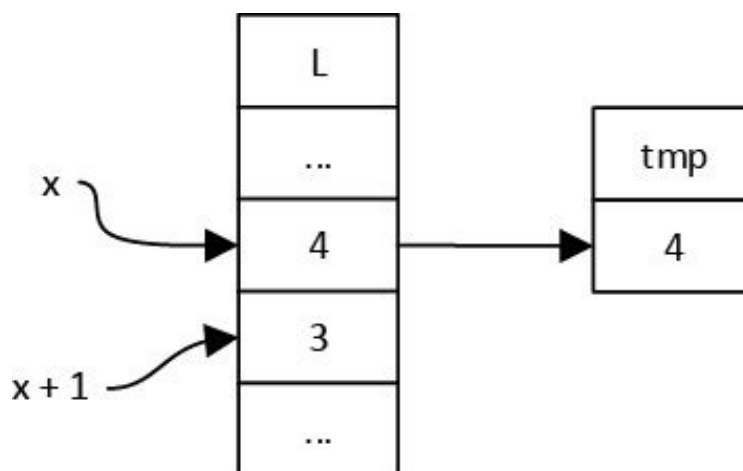


Figura 6.3 – Cambio paso a paso. Primera etapa.

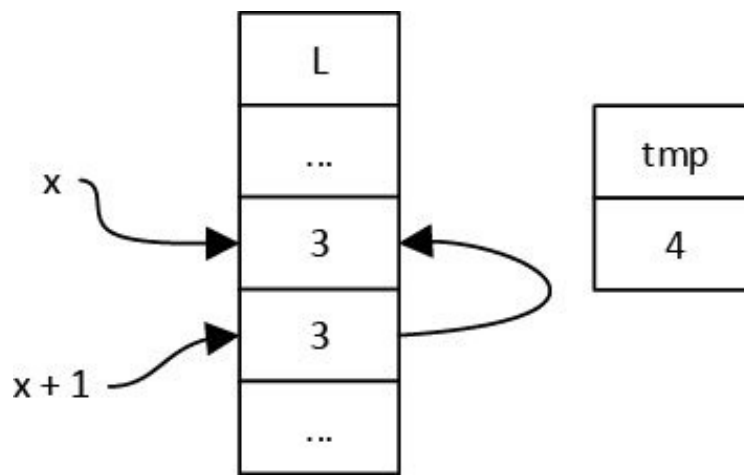


Figura 6.4 – Cambio paso a paso. Segunda etapa.

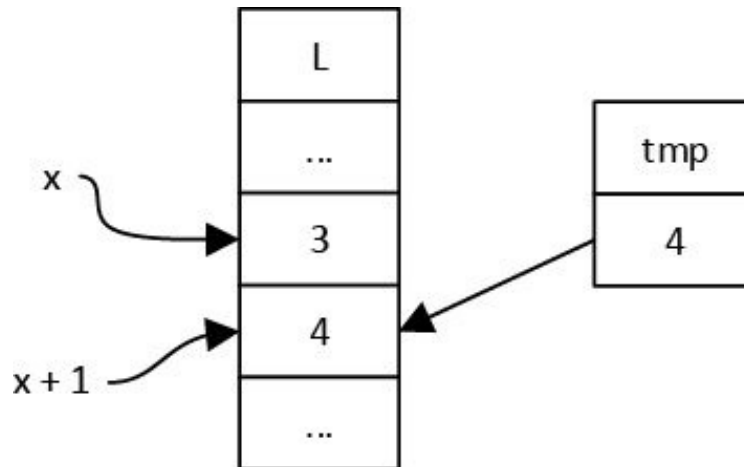


Figura 6.5 – Cambio paso a paso. Tercera etapa.

Puede entender mejor el proceso de cambio de valores utilizando un problema del día a día: imagine que usted tiene dos tazas, una con leche y otra con café. El problema consiste en pasar la leche a la taza que contiene el café, y el café a la taza que contiene la leche: usted debe intercambiar el contenido de las tazas. En este caso, resolveríamos el problema utilizando una tercera taza para facilitar la operación. Hicimos lo mismo durante el ordenamiento, donde llamamos **temp** a nuestra tercera variable.

Cuando la repetición iniciada en ❶ termina, el mayor elemento está posicionado en la última posición de la lista, que es su posición correcta. En ❸ verificamos si algo fue cambiado en la repetición anterior. Si no cambiamos nada de lugar, nuestra lista está ordenada y no necesitamos ejecutar la repetición otra vez, por eso el **break**. En caso contrario, como la última posición ya está con el elemento correcto, disminuiríamos el valor de **fin** ❹ para que no sea necesario volver a verificarlo.

Vamos a rastrear el algoritmo y a observar cómo funciona todo eso. Observe el rastreo completo en la tabla 6.1. La columna línea indica la línea del programa que está siendo ejecutada: no confundir con los números dentro de los círculos blancos de la lista.

Ejercicio 6.14 ¿Qué sucede cuando la lista ya está ordenada? Rastree el programa de la lista 6.44, pero con la lista $L=[1,2,3,4,5]$.

Ejercicio 6.15 ¿Qué sucede cuando dos valores son iguales? Rastree el programa de la lista 6.44, pero con la lista

L=[3,3,1,5,4].

Ejercicio 6.16 Modifique el programa de la lista 6.44 para ordenar la lista en orden decreciente. L=[1,2,3,4,5] debe ser ordenada como L=[5,4,3,2,1].

Tabla 6.1 – Rastreo del ordenamiento con Bubble Sort

Línea	L [0]	L [1]	L [2]	L [3]	L [4]	fin	cambió	x	temp
1	7	4	3	12	8				
2						5			
4							False		
5								0	
8							True		
9									7
10	4								
11		7							
12								1	
8							True		
9									7
10		3							
11			7						
12								2	
12								3	
8							True		
9									12
10				8					
11					12				
12								4	
15						4			
4							False		
5								0	
8							True		
9									4
10	3								
11		4							
12								1	
12								2	
12								3	
15						3			
4							False		
5								0	
12								1	
12								2	

6.17 Diccionarios

Los diccionarios consisten en una estructura de datos similar a las listas, pero con propiedades de acceso diferentes. Un diccionario está compuesto por un conjunto de claves y valores. El diccionario en sí consiste en relacionar una clave a un valor específico.

En Python, creamos diccionarios utilizando claves ({}). Cada elemento del diccionario es una combinación de clave y valor. Veamos un ejemplo donde los precios de mercaderías sean como los de la tabla 6.2.

Tabla 6.2 – Precios de mercaderías

Producto	Precio
Lechuga	€ 0,45
Batata	€ 1,20
Tomate	€ 2,30
Poroto	€ 1,50

La tabla 6.2 puede ser vista como un diccionario, donde clave sería el producto; y valor, su precio. Veamos cómo crear ese diccionario en Python en la lista 6.45.

► **Lista 6.45 – Creación de un diccionario**

```
tabla = {"Lechuga": 0.45,
        "Batata": 1.20,
        "Tomate": 2.30,
        "Poroto": 1.50 }
```

Un diccionario es accedido por sus claves. Para obtener el precio de la lechuga, digite en el intérprete, después de haber creado la tabla, `tabla["Lechuga"]`; donde `tabla` es el nombre de la variable del tipo diccionario, y “Lechuga” es nuestra clave. El valor retornado es el mismo que asociamos en la tabla, o sea, 0.45.

A diferencia de las listas, donde el índice es un número, los diccionarios utilizan sus claves como índice. Cuando atribuimos un valor a una clave, dos cosas pueden ocurrir:

- 1. Si la clave ya existe: el valor asociado es cambiado por el nuevo valor.
- 2. Si la clave no existe: la nueva clave será agregada al diccionario.

Observe la lista 6.46. En ❶, accedemos al valor asociado a la clave “Tomate”. En ❷, alteramos el valor asociado a la clave “Tomate” por un nuevo valor. Observe que el valor anterior se perdió. En ❸ creamos una nueva clave, “Cebolla”, que es adicionada al diccionario. Vea también cómo Python imprime el diccionario. Otra diferencia entre diccionarios y listas es que, al utilizar diccionarios, perdemos la noción de orden. Observe que durante la manipulación del diccionario, el orden de las claves fue alterado.

► **Lista 6.46 – Funcionamiento del diccionario**

```
>>> tabla = {"Lechuga": 0.45,
...         "Batata": 1.20,
...         "Tomate": 2.30,
...         "Poroto": 1.50}
>>> print(tabla["Tomate"]) ❶
2.3
```

```
>>> print(tabla)
{'Batata': 1.2, 'Lechuga': 0.45, 'Tomate': 2.3, 'Poroto': 1.5}
>>> tabla["Tomate"] = 2.50 ❷
>>> print(tabla["Tomate"])
2.5
>>> tabla["Cebolla"] = 1.20 ❸
>>> print(tabla)
{'Batata': 1.2, 'Lechuga': 0.45, 'Tomate': 2.5, 'Cebolla': 1.2, 'Poroto': 1.5}
```

En cuanto al acceso a los datos, tenemos que verificar si una clave existe, antes de accederla (Lista 6.47).

► Lista 6.47 – Acceso a una clave inexistente

```
>>> tabla = {"Lechuga": 0.45,
...          "Batata": 1.20,
...          "Tomate": 2.30,
...          "Poroto": 1.50}
>>> print(tabla["Manga"])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Manga'
```

Si la clave no existe, una excepción del tipo `KeyError` será activada. Para verificar si una clave pertenece al diccionario, podemos usar el operador `in` (Lista 6.48).

► Lista 6.48 – Verificación de la existencia de una clave

```
>>> tabla = {"Lechuga": 0.45,
...          "Batata": 1.20,
...          "Tomate": 2.30,
...          "Poroto": 1.50 }
>>> print("Manga" in tabla)
False
>>> print("Batata" in tabla)
True
```

Podemos también obtener una lista con las claves del diccionario, o aún una lista de los valores asociados (Lista 6.49).

► Lista 6.49 – Obtención de una lista de claves y valores

```
>>> tabla = {"Lechuga": 0.45,
...          "Batata": 1.20,
...          "Tomate": 2.30,
```



```
...         "Poroto": 1.50}
>>> print(tabla.keys())
dict_keys(['Batata', 'Lechuga', 'Tomate', 'Poroto'])
>>> print(tabla.values())
dict_values([1.2, 0.45, 2.3, 1.5])
```

Observe que los métodos `keys()` y `values()` retornan generadores. Puede utilizarlos directamente dentro de un `for` o transformarlos en lista usando la función `list`.

Veamos un programa que utiliza diccionarios para exhibir el precio de un producto en la lista 6.50.

► Lista 6.50 – Obtención del precio con un diccionario

```
tabla = {"Lechuga": 0.45,
        "Batata": 1.20,
        "Tomate": 2.30,
        "Poroto": 1.50}

while True:
    producto = input("Digite el nombre del producto, fin para terminar:")
    if producto == "fin":
        break
    if producto in tabla: ❶
        print("Precio %5.2f" % tabla[producto]) ❷
    else:
        print(";Producto no encontrado!")
```

En ❶ verificamos si el diccionario contiene la clave buscada. En caso afirmativo, imprimimos el precio asociado a la clave, o habrá un mensaje de error.

Para borrar una clave, utilizaremos la instrucción `del` (Lista 6.51).

► Lista 6.51 – Exclusión de una asociación del diccionario

```
>>> tabla = {"Lechuga": 0.45,
...         "Batata": 1.20,
...         "Tomate": 2.30,
...         "Poroto": 1.50}
>>> del tabla["Tomate"]
>>> print(tabla)
{'Batata': 1.2, 'Lechuga': 0.45, 'Poroto': 1.5}
```

Usted puede estar preguntándose cuándo utilizar listas y cuando utilizar diccionarios. Todo depende de lo que desee realizar. Si puede acceder fácilmente a sus datos por sus claves, casi nunca necesitará acceder a ellos de una sola vez: un diccionario es más interesante. Además de eso, puede acceder a los valores asociados a una clave rápidamente sin investigar. La implementación interna de diccionarios también garantiza una buena velocidad de acceso cuando tenemos muchas claves. Sin

embargo, un diccionario no organiza sus claves, dicho de otro modo: las primeras claves introducidas no siempre serán las primeras en la lista de claves. Si sus datos necesitan preservar el orden de inserción (como en colas o pilas, continúe usando listas), los diccionarios no serán una opción.

6.18 Diccionarios con listas

En Python, podemos tener diccionarios en los cuales las claves están asociadas a listas o a otros diccionarios. Imagine una relación de stock de mercaderías donde tendríamos, además del precio, la cantidad de mercadería en stock (Lista 6.52).

► Lista 6.52 – Diccionario con listas

```
stock = {"tomate": [ 1000, 2.30],
        "lechuga": [500, 0.45],
        "batata": [2001, 1.20],
        "poroto": [100, 1.50]}
```

En ese caso, el nombre del producto es la clave, y la lista consiste en los valores asociados, una lista por clave. El primer elemento de la lista es la cantidad disponible; y el segundo es el precio del producto.

Una aplicación sería procesar una lista de operaciones y calcular el precio total de venta, actualizando también la cantidad en stock.

► Lista 6.53 – Ejemplo de diccionario con stock y operaciones de venta

```
stock={"tomate": [ 1000, 2.30],
      "lechuga": [500, 0.45],
      "batata": [2001, 1.20],
      "poroto": [100, 1.50]}
venta = [{"tomate", 5}, {"batata", 10}, {"lechuga",5}]
total = 0
print("Ventas:\n")
for operación in venta:
    producto, cantidad = operación ❶
    precio = stock[producto][1] ❷
    costo = precio * cantidad
    print("%12s: %3d x %6.2f = %6.2f" % (producto, cantidad, precio, costo))
    stock[producto][0] -= cantidad ❸
    total += costo
print(" Costo total: %21.2f\n" % total)
print("Stock:\n")
for clave, datos in stock.items(): ❹
    print("Descripción: ", clave)
```

```
print("Cantidad: ", datos[0])
print("Precio: %6.2f\n" % datos[1])
```

En ❶ utilizamos una operación de desempaquetamiento, como ya hicimos con `for` y `enumerate`. Como **operación** es una lista con dos elementos, al escribir **producto**, **cantidad** tenemos el primer elemento de **operación** atribuido a **producto**; y el segundo a **cantidad**. De esa forma, la construcción es equivalente a:

```
producto = operación[0]
cantidad = operación[1]
```

En ❷, utilizamos el contenido de **producto** como clave en el diccionario **stock**. Como nuestros datos son una lista, elegimos el segundo elemento, que almacena el precio del referido producto. Observe que atribuir nombres a cada uno de esos componentes facilita la lectura del programa.

Imagine escribir:

```
precio = stock[operación[0]][1]
```

En ❸, actualizamos la cantidad en stock sustrayendo la cantidad vendida del stock actual.

Ya en ❹ utilizamos el método **ítems** del objeto diccionario. El método **ítems** retorna una tupla conteniendo la clave y el valor de cada ítem almacenado en el diccionario. Usando un **for** con dos variables, **clave** y **datos**, efectuamos el desempaquetamiento de esos valores en un solo pasaje. Para entender mejor cómo sucede eso, pruebe alterar el programa para exhibir el valor de clave y datos en cada iteración.

Ejercicio 6.17 Altere el programa de la lista 6.53 de modo de solicitar al usuario el producto y la cantidad vendida. Verifique si el nombre del producto digitado existe en el diccionario, y solo entonces efectúe la baja en stock.

Ejercicio 6.18 Escriba un programa que genere un diccionario donde cada clave sea un carácter, y su valor sea el número de ese carácter encontrado en una frase leída.

Ejemplo: El rato -> {"E":1, "r":1, "a":1, "t":1, "o":1, "l": 1}

6.19 Tuplas

Las tuplas pueden ser vistas como listas en Python, con la gran diferencia que son inmutables. Las tuplas son ideales para representar listas de valores constantes y también para realizar operaciones de empaquetamiento y desempaquetamiento de valores. Primero veamos cómo crear una tupla.

Las tuplas son creadas de forma semejante a las listas, pero utilizamos paréntesis en vez de corchetes. Por ejemplo:

```
>>> tupla = ("a", "b", "c")
>>> tupla
('a', 'b', 'c')
```

Las tuplas soportan la mayor parte de las operaciones de lista, como rebanado e indexación.

```

>>> tupla[0]
'a'
>>> tupla[2]
'c'
>>> tupla[1:]
('b', 'c')
>>> tupla * 2
('a', 'b', 'c', 'a', 'b', 'c')
>>> len(tupla)
3

```

Pero las tuplas no pueden tener sus elementos alterados. Vea qué sucede si tratamos de alterar una tupla:

```

>>> tupla[0]="a"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment

```

Varias funciones utilizan o generan tuplas en Python. Las tuplas pueden ser utilizadas con **for**:

```

>>> for elemento in tupla:
...     print(elemento)
...
a
b
c

```

Python también permite crear tuplas usando valores separados por coma, independientemente de usar paréntesis:

```

>>> tupla = 100, 200, 300
>>> tupla
(100, 200, 300)

```

En este caso, 100, 200 y 300 fueron convertidos en una tupla con tres elementos. Ese tipo de operación es llamado de empaquetamiento.

Las tuplas también pueden ser utilizadas para desempaquetar valores, por ejemplo:

```

>>> a, b = 10, 20
>>> a
10
>>> b
20

```

Donde el primer valor, 10, fue atribuido a la primera variable **a** y 20 a la segunda, **b**. Ese tipo de construcción es interesante para distribuir el valor de una tupla en varias variables.

También podemos cambiar rápidamente los valores de variables con construcciones del tipo:

```
>>> a, b = 10, 20
>>> a, b = b, a
>>> a
20
>>> b
10
```

Donde la tupla de la izquierda fue usada para atribuir los valores a la derecha. En ese caso, las atribuciones `a = b` y `b = a` fueron realizadas inmediatamente, sin necesidad de utilizar una variable intermedia para el cambio.

La sintaxis de Python es un tanto especial cuando necesitamos crear tuplas con un único elemento. Como los valores se escriben entre paréntesis, cuando solo un valor está presente, debemos agregar una coma para indicar que el valor es una tupla con un único elemento. Vea lo que sucede, usando y no usando la coma:

```
>>> t1 = (1)
>>> t1
1
>>> t2 = (1,)
>>> t2
(1,)
>>> t3 = 1,
>>> t3
(1,)
```

Observe que en `t1` no utilizamos la coma y el código fue interpretado como un número entero entre paréntesis. En `t2` utilizamos la coma, y nuestra tupla fue correctamente construida. En `t3`, creamos otra tupla, pero en ese caso no necesitamos usar paréntesis.

Podemos también crear tuplas vacías, escribiendo solo los paréntesis:

```
>>> t4 = ()
>>> t4
()
>>> len(t4)
0
```

Las tuplas también pueden ser creadas a partir de listas, utilizando la función `tuple`:

```
>>> L = [1, 2, 3]
>>> T = tuple(L)
>>> T
(1, 2, 3)
```

Aunque no podamos alterar una tupla después de su creación, podemos concatenarlas, generando

nuevas tuplas:

```
>>> t1 = (1, 2, 3)
>>> t2 = (4, 5, 6)
>>> t1 + t2
(1, 2, 3, 4, 5, 6)
```

Observe que si una tupla contiene una lista u otro objeto que puede ser alterado, el mismo continuará funcionando normalmente. Vea el ejemplo de una tupla que contiene una lista:

```
>>> tupla=("a", ["b", "c", "d"])
>>> tupla
('a', ['b', 'c', 'd'])
>>> len(tupla)
2
>>> tupla[1]
['b', 'c', 'd']
>>> tupla[1].append("e")
>>> tupla
('a', ['b', 'c', 'd', 'e'])
```

En este caso, nada cambió en la tupla en sí, sino en la lista que es su segundo elemento. O sea, la tupla no fue alterada, pero la lista que ella contenía, sí.

Trabajando con cadenas de caracteres

En el capítulo 3 vimos que podemos acceder a cadenas de caracteres como listas, pero también vimos que las cadenas de caracteres son inmutables en Python. Veamos lo que sucede en la lista 7.1.

► Lista 7.1 – Alteración de una cadena de caracteres

```
>>> S = "Hola mundo"
>>> print(S[0])
A
>>> S[0] = "a"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Si queremos trabajar carácter por carácter con una cadena de caracteres, alterando su valor, tendremos primero que transformarla en una lista (Lista 7.2).

► Lista 7.2 – Convirtiendo una cadena de caracteres en lista

```
>>> L = list("Hola Mundo")
>>> L[0] = "a"
>>> print(L)
['H', 'o', 'l', 'a', ' ', 'M', 'u', 'n', 'd', 'o']
>>> s="".join(L)
>>> print(s)
Hola Mundo
```

La función `list` transforma cada carácter de la cadena de caracteres en un elemento de la lista retornada. Ya el método `join` hace la operación inversa, transformando los elementos de la lista en cadena de caracteres.

7.1 Verificación parcial de cadenas de caracteres

Cuando necesite verificar si una cadena de caracteres comienza o termina con algunos caracteres, puede usar los métodos `startswith` y `endswith`. Esos métodos verifican solo los primeros (`startswith`) o los últimos (`endswith`) caracteres de la cadena de caracteres, retornando `True` en caso que sean iguales o `False` en caso contrario.

► Lista 7.3 – Verificación parcial de cadenas de caracteres

```
>>> nombre = "Juan da Silva"
>>> nombre.startswith("Juan")
True
>>> nombre.startswith("juan")
False
>>> nombre.endswith("Silva")
True
```

Observe la lista 7.3. Vea que comparamos “Juan da Silva” con “juan” y obtuvimos **False**. Ese es un detalle pequeño, pero importante, pues **startswith** y **endswith** consideran letras mayúsculas y minúsculas como letras diferentes.

Puede resolver ese tipo de problema convirtiendo la cadena de caracteres para mayúsculas o minúsculas antes de realizar la comparación. El método **lower** retorna una copia de la cadena de caracteres con todos los caracteres en minúscula, y el método **upper** retorna una copia con todos los caracteres en mayúscula. Vea los ejemplos en la lista 7.4.

► Lista 7.4 – Ejemplos de conversión en mayúsculas y minúsculas

```
>>> s="Erre con erre guitarra"
>>> s.lower()
'erre con erre guitarra'
>>> s.upper()
'ERRE CON ERRE GUITARRA'
>>> s.lower().startswith("erre")
True
>>> s.upper().startswith("ERRE")
True
```

No se olvide de comparar con una cadena de caracteres donde todos los caracteres estén en mayúscula o minúscula, dependiendo de si utilizó **upper** o **lower**, respectivamente.

Otra forma de verificar si una palabra pertenece a una cadena de caracteres es utilizando el operador **in**. Veamos los ejemplos de la lista 7.5.

► Lista 7.5 – Investigación de palabras en una cadena de caracteres usando in

```
>>> s = "María Amélia Souza"
>>> "Amélia" in s
True
>>> "María" in s
True
>>> "Souza" in s
True
```



```
>>> "a A" in s
```

```
True
```

```
>>> "amélia" in s
```

```
False
```

Usted también puede probar si una cadena de caracteres no está contenida en otra, utilizando `not in` (Lista 7.6).

► Lista 7.6 – Investigación de palabras en una cadena de caracteres usando `not in`

```
>>> s = "Todos los caminos llevan a Roma"
```

```
>>> "llevan" not in s
```

```
False
```

```
>>> "Caminos" not in s
```

```
True
```

```
>>> "AS" not in s
```

```
True
```

Vea que aquí también las letras mayúsculas y minúsculas son diferentes. Puede combinar `lower` y `upper` con `in` y `not in` para ignorar ese tipo de diferencia, como muestra la lista 7.7.

► Lista 7.7 – Combinación de `lower` y `upper` con `in` y `not in`

```
>>> s="Juan compró un auto"
```

```
>>> "juan" in s.lower()
```

```
True
```

```
>>> "AUTO" in s.upper()
```

```
True
```

```
>>> "compró" not in s.lower()
```

```
False
```

```
>>> "barco" not in s.lower()
```

```
True
```

El operador `in` también puede ser utilizado con listas normales, facilitando así la investigación de elementos dentro de una lista.

7.2 Recuento

Si Usted necesitar contar las ocurrencias de una letra o palabra en una cadena de caracteres, utilice el método `count`. Vea el ejemplo en la lista 7.8.

► Lista 7.8 – Recuento de letras y palabras

```
>>> t = "un tigre, dos tigres, tres tigres"
```

```
>>> t.count("tigre")
```

```
3
```

```
>>> t.count("tigres")
2
>>> t.count("t")
4
>>> t.count("z")
0
```

7.3 Investigación de cadenas de caracteres

Para investigar si una cadena de caracteres está dentro de otra y obtener la posición de la primera ocurrencia, puede utilizar el método `find` (Lista 7.9).

► Lista 7.9 – Investigación de cadenas de caracteres con `find`

```
>>> s="Hola mundo"
>>> s.find("mun")
5
>>> s.find("ok")
-1
```

En caso que la cadena de caracteres sea encontrada, usted obtendrá un valor mayor o igual a cero, o -1, en caso contrario. Observe que el valor retornado, cuando es mayor o igual a cero, es igual al índice que puede ser utilizado para obtener el primer carácter de la cadena de caracteres buscada.

Si el objetivo es investigar, pero de derecha a izquierda, utilice el método `rfind`, que realiza esa tarea (Lista 7.10).

► Lista 7.10 – Investigación de cadenas de caracteres con `rfind`

```
>>> s="Un día de sol"
>>> s.rfind("d")
7
>>> s.find("d")
3
```

Tanto `find` cuanto `rfind` soportan dos opciones adicionales: comienzo (*start*) y fin (*end*). Si especifica comienzo, la investigación empezará a partir de esa posición. Si especifica fin, la investigación utilizará esa posición como último carácter a considerar en la investigación. Vea los ejemplos en la lista 7.11.

► Lista 7.11 – Investigación de cadenas de caracteres, limitando el comienzo o el fin

```
>>> s = "un tigre, dos tigres, tres tigres"
>>> s.find("tigres")
14
>>> s.rfind("tigres")
27
```

```
>>> s.find("tigres",7) #comienzo=7
14
>>> s.find("tigres",30) #comienzo=30
-1
>>> s.find("tigres",0,10) #comienzo=0 fin=10
-1
```

Podemos usar el valor retornado por `find` y `rfind` para encontrar todas las ocurrencias de la cadena de caracteres. Por ejemplo, el programa de la lista 7.12 produce la salida de la lista 7.13.

► Lista 7.12 – Investigación de todas las ocurrencias

```
s = "un tigre, dos tigres, tres tigres"
p = 0
while p > -1:
    p = s.find("tigre", p)
    if p >= 0:
        print("Posición: %d" % p)
        p+=1
```

► Lista 7.13 – Resultado de la investigación

```
Posición: 3
Posición: 14
Posición: 27
```

Los métodos `index` y `rindex` son bien parecidos a `find` y `rfind`, respectivamente. La mayor diferencia es que si la substring no es encontrada, `index` y `rindex` lanzan una excepción del tipo `ValueError`.

Ejercicio 7.1 Escriba un programa que lea dos cadenas de caracteres. Verifique si la segunda ocurre dentro de la primera e imprima la posición de comienzo.

1ª cadena de caracteres: AABBEFAATT

2ª cadena de caracteres: BE

Resultado: BE encontrado en la posición 3 de AABBEFAATT

Ejercicio 7.2 Escriba un programa que lea dos cadenas de caracteres y genere una tercera con los caracteres comunes a las dos cadenas de caracteres leídas.

1ª cadena de caracteres: AAACCTBF

2ª cadena de caracteres: CBT

Resultado: CBT

El orden de los caracteres de la cadena de caracteres generada no es importante, pero debe contener todas las letras comunes a ambas.

Ejercicio 7.3 Escriba un programa que lea dos cadenas de caracteres y genere una tercera solo con los caracteres que aparecen en una de ellas.

1ª cadena de caracteres: CTA

2ª cadena de caracteres: ABC

3ª cadena de caracteres: BT

El orden de los caracteres de la tercera cadena de caracteres no es importante.

Ejercicio 7.4 Escriba un programa que lea una cadena de caracteres e imprima cuántas veces cada carácter aparece en esa cadena de caracteres.

Cadena de caracteres: TTAAC

Resultado:

T: 2x

A: 2x

C: 1x

Ejercicio 7.5 Escriba un programa que lea dos cadenas de caracteres y genere una tercera, en la cual los caracteres de la segunda hayan sido retirados de la primera.

1ª cadena de caracteres: AATTGGAA

2ª cadena de caracteres: TG

3ª cadena de caracteres: AAAA

Ejercicio 7.6 Escriba un programa que lea tres cadenas de caracteres. Imprima el resultado de la sustitución en la primera, de los caracteres de la segunda por los de la tercera.

1ª cadena de caracteres: AATTTCGAA

2ª cadena de caracteres: TG

3ª cadena de caracteres: AC

Resultado: AAAACCAA

7.4 Posicionamiento de cadenas de caracteres

Python también trae métodos que ayudan a presentar cadenas de caracteres de formas más interesantes. Veamos el método `center`, que centraliza la cadena de caracteres en un número de posiciones pasado como parámetro, llenando con espacios a la derecha y a la izquierda hasta que la cadena de caracteres esté centralizada (Lista 7.14).

► Lista 7.14 – Centralización de texto en una cadena de caracteres

```
>>> s="tigre"
>>> print("X"+s.center(10)+"X")
X  tigre  X
>>> print("X"+s.center(10,".")+"X")
X..tigre...X
```

Si, además del tamaño, usted también pasa el carácter de llenado, este será utilizado en el lugar de espacios en blanco.

Si lo que desea es solo completar la cadena de caracteres con espacios a la izquierda, puede utilizar el método `ljust`. Si desea completar con espacios a la derecha, utilice `rjust` (Lista 7.15).

► Lista 7.15 – Llenado de cadenas de caracteres con espacios

```
>>> s="tigre"
>>> s.ljust(20)
'tigre                '
>>> s.rjust(20)
'                tigre'
>>> s.ljust(20,".")
'tigre.....'
>>> s.rjust(20,"-")
'-----tigre'
```

Esas funciones son útiles cuando necesitamos crear informes o simplemente alinear la salida de los programas.

7.5 Separación de cadenas de caracteres

El método `split` separa una cadena de caracteres a partir de un carácter pasado como parámetro, retornando una lista con las substrings ya separadas. Vea un ejemplo en la lista 7.16.

► Lista 7.16 – Separación de cadenas de caracteres

```
>>> s="un tigre, dos tigres, tres tigres"
>>> s.split(",")
['un tigre', ' dos tigres', ' tres tigres']
>>> s.split(" ")
['un', 'tigre,', 'dos', 'tigres,', 'tres', 'tigres']
>>> s.split()
['un', 'tigre,', 'dos', 'tigres,', 'tres', 'tigres']
```

Observe que el carácter que utilizamos para dividir la cadena de caracteres no es retornado en la lista, o sea, el mismo es utilizado para separar la cadena de caracteres y después es descartado.

Si desea separar una cadena de caracteres con varias líneas de texto, puede utilizar el método `splitlines` (Lista 7.17).

► Lista 7.17 – Separación de cadenas de caracteres de varias líneas

```
>>> m="Una línea\notra línea\ny más una\n"
>>> m.splitlines()
['Una línea', 'otra línea', 'y más una']
```

7.6 Sustitución de cadenas de caracteres

Para sustituir tramos de una cadena de caracteres por otros, utilice el método **replace**. Con el método **replace**, el primer parámetro es la cadena de caracteres a sustituir; y el segundo, el contenido que la sustituirá. Opcionalmente, podemos pasar un tercer parámetro que limita cuántas veces queremos realizar la repetición. Veamos algunos ejemplos en la lista 7.18.

► Lista 7.18 – Sustitución de cadenas de caracteres

```
>>> s="un tigre, dos tigres, tres tigres"
>>> s.replace("tigre", "gato")
'un gato, dos gatos, tres gatos'
>>> s.replace("tigre", "gato", 1)
'un gato, dos tigres, tres tigres'
>>> s.replace("tigre", "gato", 2)
'un gato, dos gatos, tres tigres'
>>> s.replace("tigre", "")
'un , dos s, tres s'
>>> s.replace("", "-")
'-u-n- -t-i-g-r-e-, -d-o-s- -t-i-g-r-e-s-, -t-r-e-s- -t-i-g-r-e-s-'
```

Si usted pasa una cadena de caracteres vacía en el segundo parámetro, el tramo será borrado. Si el primer parámetro es una cadena de caracteres vacía, el segundo será introducido antes de cada carácter de la cadena de caracteres.

7.7 Remoción de espacios en blanco

El método **strip** es utilizado para remover espacios en blanco del comienzo o del final de la cadena de caracteres. Los métodos **lstrip** y **rstrip** remueven solo los caracteres en blanco a la izquierda o a la derecha, respectivamente (Lista 7.19).

► Lista 7.19 – Remoción de espacios en blanco con strip, lstrip y rstrip

```
>>> t="   Hola   "
>>> t.strip()
'Hola'
>>> t.lstrip()
'Hola   '
>>> t.rstrip()
'   Hola'
```

Si pasa un parámetro tanto para **strip** como para **lstrip** o **rstrip**, este será utilizado como carácter a remover (Lista 7.20).

► Lista 7.20 – Remoción de caracteres con strip, lstrip y rstrip

```
>>> s="...///Hola///..."
>>> s.lstrip(".")
'///Hola///...'
>>> s.rstrip(".")
'...///Hola///'
>>> s.strip(".")
'///Hola///'
>>> s.strip("./")
'Hola'
```

7.8 Validación por tipo de contenido

El contenido de las cadenas de caracteres en Python puede ser analizado y verificado utilizando métodos especiales. Esos métodos verifican si todos los caracteres son letras, números o una combinación de ellos. Veamos algunos ejemplos en la lista 7.21.

► Lista 7.21 – Validación de cadenas de caracteres por su contenido

```
>>> s = "125"
>>> p = "hola mundo"
>>> s.isalnum()
True
>>> p.isalnum()
False
>>> s.isalpha()
False
>>> p.isalpha()
False
```

El método `isalnum` retorna verdadero si la cadena de caracteres no está vacía, y si todos sus caracteres son letras y/o números. Si la cadena de caracteres contiene otros tipos de caracteres, como espacios, coma, exclamación, interrogación o caracteres de control, retorna **False**.

El método `isalpha` es más restrictivo, retornando verdadero solo si todos los caracteres son letras, incluyendo vocales acentuadas. Retorna falso si algún otro tipo de carácter es encontrado en la cadena de caracteres o si está vacía.

El método `isdigit` verifica si el valor consiste en números, retornando **True** si la cadena de caracteres no está vacía y contiene solo números. Si la cadena de caracteres contiene espacios, puntos, comas o señales (+ o -), retorna falso (Lista 7.22).

► Lista 7.22 – Validación de cadenas de caracteres con números

```
>>> "771".isdigit()
True
>>> "10.4".isdigit()
```

False

```
>>> "+10".isdigit()
```

False

```
>>> "-5".isdigit()
```

False

Los métodos `isdigit` e `isnumeric` son parecidos, y diferenciarlos supone un conocimiento de Unicode más profundo. `isdigit` retorna `True` para caracteres definidos como dígitos numéricos en Unicode, yendo más allá de nuestros 0 a 9; como, por ejemplo, un 9 tibetano (`\u0f29`). `isnumeric` es más abarcador, incluyendo dígitos y representaciones numéricas como fracciones; por ejemplo $1/3$ (`\u2153`). Veamos algunas pruebas en la lista 7.23.

► Lista 7.23 – Diferenciación de `isnumeric` de `isdigit`

```
>>> untercio="\u2153"
```

```
>>> nuevetibetano="\u0f29"
```

```
>>> untercio.isdigit()
```

False

```
>>> untercio.isnumeric()
```

True

```
>>> nuevetibetano.isdigit()
```

True

```
>>> nuevetibetano.isnumeric()
```

True

Podemos también verificar si todos los caracteres de una cadena de caracteres son letras mayúsculas o minúsculas usando `isupper` y `islower`, respectivamente (Lista 7.24).

► Lista 7.24 – Verificación de mayúsculas y minúsculas

```
>>> s = "ABC"
```

```
>>> p = "abc"
```

```
>>> e = "aBc"
```

```
>>> s.isupper()
```

True

```
>>> s.islower()
```

False

```
>>> p.isupper()
```

False

```
>>> p.islower()
```

True

```
>>> e.isupper()
```

False


```
>>> e.islower()
```

False

Tenemos también cómo verificar si la cadena de caracteres contiene solo caracteres en blanco; como espacios, marcas de tabulación (TAB), quiebres de línea (LF) o retornos de carro (CR). Para eso, vamos a utilizar el método `isspace`, como muestra la lista 7.25.

► Lista 7.25 – Verificación si la cadena de caracteres contiene solo caracteres de espaciamento

```
>>> "\t\n\r".isspace()
```

True

```
>>> "\tHola".isspace()
```

False

Si necesita verificar si algo puede ser impreso en la pantalla, el método `isprintable` puede ayudar. Este método retorna **False** si encuentra algún carácter que no puede ser impreso, en la cadena de caracteres. Puede utilizarlo para verificar si la impresión de una cadena de caracteres puede causar efectos indeseados en la terminal o en el formateo de un archivo (Lista 7.26).

► Lista 7.26 – Verificar si la cadena de caracteres puede ser impresa

```
>>> "\n\t".isprintable()
```

False

```
>>> "\nHola".isprintable()
```

False

```
>>> "Hola mundo".isprintable()
```

True

7.9 Formateo de cadenas de caracteres

La versión 3 de Python introdujo una nueva forma de representar máscaras en cadenas de caracteres. Esa nueva forma es más poderosa que las tradicionales máscaras que utilizamos, combinando `%d`, `%s`, `%f`.

La nueva forma representa los valores a sustituir, entre claves. Veamos un ejemplo en la lista 7.27.

► Lista 7.27 – Formateo de cadenas de caracteres con el método `format`

```
>>> "{0} {1}".format("Hola", "Mundo")
```

```
'Hola Mundo'
```

```
>>> "{0} x {1} €{2}".format(5, "manzana", "1.20")
```

```
'5 x manzana €1.20'
```

El número entre corchetes es una referencia a los parámetros pasados al método `format`, donde 0 es el primer parámetro; 1, el segundo; y así sucesivamente, como los índices de una lista. Una de las ventajas de la nueva sintaxis es poder utilizar el mismo parámetro varias veces en la cadena de caracteres (Lista 7.28).

► Lista 7.28 – Uso del mismo parámetro más de una vez

```
>>> "{0} {1} {0}".format("-", "x")
'- x -'
```

Eso también permite el completo reordenamiento del mensaje, tal como imprimir los parámetros en otro orden (Lista 7.29).

► Lista 7.29 – Alteración del orden de utilización de los parámetros

```
>>> "{1} {0}".format("primero", "segundo")
'segundo primero'
```

La nueva sintaxis permite también especificar el ancho de cada valor, utilizando el símbolo de dos puntos (:) después de la posición del parámetro, como 0:10 de la lista 7.30, que significa: sustituya el primer parámetro con un ancho de 10 caracteres. Si el primer parámetro es menor que el tamaño informado, serán utilizados espacios para completar las posiciones que faltan. Si el parámetro es mayor que el tamaño especificado, será impreso en su totalidad, ocupando más espacio que el inicialmente especificado.

► Lista 7.30 – Limitación del tamaño de impresión de los parámetros

```
>>> "{0:10} {1}".format("123", "456")
'123          456'
>>> "X{0:10}X".format("123")
'X123          X'
>>> "X{0:10}X".format("123456789012345")
'X123456789012345X'
```

Podemos también especificar si queremos los espacios adicionales a la izquierda o a la derecha del valor, utilizando los símbolos de mayor (>) y menor (<) luego de los dos puntos (Lista 7.31).

► Lista 7.31 – Especificación de espacios a la izquierda o a la derecha

```
>>> "X{0:<10}X".format("123")
'X123          X'
>>> "X{0:>10}X".format("123")
'X          123X'
```

Si queremos el valor entre los espacios, de modo de centrarlo, podemos utilizar el acento circunflejo (^), como muestra la lista 7.32.

► Lista 7.32 – Centrado

```
>>> "X{0:^10}X".format("123")
'X    123    X'
```

Si queremos otro carácter en el lugar de espacios, podemos especificarlo después de los dos puntos (Lista 7.33).

► Lista 7.33 – Especificación de espacios a la izquierda o a la derecha

```
>>> "X{0:.<10}X".format("123")
'X123.....X'
>>> "X{0:!!>10}X".format("123")
'X!!!!!!!!123X'
>>> "X{0:*^10}X".format("123")
'X***123***X'
```

Si el parámetro es una lista, podemos especificar el índice del elemento a sustituir dentro de la máscara (Lista 7.34).

► Lista 7.34 – Máscaras con elementos de una lista

```
>>> "{0[0]} {0[1]}".format(["123", "456"])
'123 456'
```

El mismo es válido para diccionarios (Lista 7.35).

► Lista 7.35 – Máscaras con elementos de un diccionario

```
>>> "{0[nombre]} {0[teléfono]}".format({"teléfono": 572, "nombre":"María"})
'María 572'
```

Observe que dentro de la cadena de caracteres escribimos nombre y teléfono entre corchetes, pero sin comillas, como normalmente haríamos al utilizar diccionarios. Esa sintaxis es especial para el método `format`.

7.9.1 Formateo de números

La nueva sintaxis también permite el formateo de números. Por ejemplo, si especifica el tamaño a imprimir con un cero a la izquierda, el valor será impreso con el ancho determinado y con ceros a la izquierda completando el tamaño (Lista 7.36).

► Lista 7.36 – Ceros a la izquierda

```
>>> "{0:05}".format(5)
'00005'
```

Podemos también utilizar otro carácter, diferente de 0, pero en ese caso debemos escribir el carácter a la izquierda del símbolo de igualdad (Lista 7.37).

► Lista 7.37 – Llenado con otros caracteres

```
>>> "{0:*=7}".format(32)
'*****32'
```

Podemos también especificar la alineación de los números que estamos imprimiendo, usando `<`, `>` y `^` (Lista 7.38).

► Lista 7.38 – Combinación de varios códigos de formateo

```
>>> "{0:*^10}".format(123)
'***123***'
>>> "{0:*<10}".format(123)
'123*****'
>>> "{0:*>10}".format(123)
'*****123'
```

Podemos también utilizar una coma para solicitar el agrupamiento por millar (Lista 7.39), y el punto para indicar la precisión de números decimales, o mejor, la cantidad de lugares después de la coma.

► **Lista 7.39 – Separación de miles**

```
>>> "{0:10,}".format(7532)
'      7,532'
>>> "{0:10.5f}".format(1500.31)
'1500.31000'
>>> "{0:10,.5f}".format(1500.31)
'1,500.31000'
```

La nueva sintaxis también permite forzar la impresión de señales o solo reservar espacio para una impresión eventual (Lista 7.40).

► **Lista 7.40 – Impresión de señales de positivo y negativo**

```
>>> "{0:+10} {1:-10}".format(5,-6)
'      +5      -6'
>>> "{0:-10} {1: 10}".format(5,-6)
'      5      -6'
>>> "{0: 10} {1:+10}".format(5,-6)
'      5      -6'
```

Cuando trabajamos con formatos numéricos, debemos indicar con una letra el formato que debe ser adoptado para la impresión. Esa letra informa cómo debemos exhibir un número. La lista completa de formatos numéricos es presentada en las tablas 7.1 y 7.2.

Tabla 7.1 – Formatos de números enteros

Código	Descripción	Ejemplo (45)
b	Binario	101101
c	Carácter	-
d	Base 10	45
n	Base 10 local	45
o	Octal	55
x	Hexadecimal con letras minúsculas	2d
X	Hexadecimal con letras mayúsculas	2D

El formato **b** imprime el número utilizando el sistema binario que, como ya vimos, es de base 2, y solo tiene 0 y 1 como dígitos. El formato **o** imprime el número utilizando el sistema octal, que es de

base 8, con dígitos de 0 a 7. El formato `c` imprime el número convirtiéndolo en carácter, utilizando la tabla Unicode. Tanto el formato `x` como el `X` imprimen los números utilizando el sistema hexadecimal, de base 16. La diferencia es que `x` utiliza letras minúsculas, y `X`, mayúsculas. Veamos otros ejemplos en la lista 7.41.

► **Lista 7.41 – Formateo de enteros**

```
>>> "{:b}".format(5678)
'1011000101110'
>>> "{:c}".format(65)
'A'
>>> "{:o}".format(5678)
'13056'
>>> "{:x}".format(5678)
'162e'
>>> "{:X}".format(5678)
'162E'
```

Los formatos `d` y `n` son parecidos. El formato `d` es semejante al que utilizamos al formatear los números con `%d`. La diferencia entre el formato `d` y el `n` es que el `n` tiene en consideración las configuraciones regionales de la máquina del usuario. Veamos algunos ejemplos en la lista 7.42. Observe que antes de configurar la máquina para el español, el resultado de `d` y `n` eran iguales. Después de la configuración regional, el formato `n` pasó a exhibir los números utilizando puntos para separar los miles. Si utiliza Windows, modifique " `es_ES.utf-8`" para "`Spanish`" en las listas 7.42 y 7.43.

Lista 7.42 – El formato `d` y el formato `n`

```
>>> "{:d}".format(5678)
'5678'
>>> "{:n}".format(5678)
'5678'
>>> import locale
>>> locale.setlocale(locale.LC_ALL,"es_ES.utf-8")
'es_ES.utf-8'
>>> "{:n}".format(5678)
'5.678'
```

Tabla 7.2 – Formatos de números decimales

Código	Descripción	Ejemplo (1.345)
e	Notación científica con e minúscula	1.345000e+00
E	Notación científica con e mayúscula	1.345000E+00
f	Decimal	1.345000
g	Genérico	1.345

G	Genérico	1.345
n	Local	1,345
%	Porcentual	134.500000%

Para números decimales también tenemos varios códigos. Al código **f** ya lo conocemos y funciona de forma semejante al que utilizamos en **%f**. El formato **n** utiliza las configuraciones regionales para imprimir el número. En español, esa configuración utiliza el punto como separador de millar y la coma como separador decimal, produciendo números más fáciles de entender. Veamos ejemplos en la lista 7.43.

► Lista 7.43 – Formateo de números decimales

```
>>> "{:f}".format(1579.543)
'1579.543000'
>>> "{:n}".format(1579.543)
'1579.54'
>>> import locale
>>> locale.setlocale(locale.LC_ALL,"es_ES.utf-8")
'es_ES.utf-8'
>>> "{:n}".format(1579.543)
'1.579,54'
```

Los formatos **e** y **E** imprimen el número utilizando notación científica. Eso quiere decir que la parte decimal va a ser sustituida por un exponente. Por ejemplo: 1004.5 en notación científica será impreso como 1.004500e+03. Para entender ese formato, debemos entender sus partes. El número a la izquierda de **e** es el que llamamos mantisa, y el número a la derecha es el exponente. La base es siempre 10 y, para recomponer el número, multiplicamos la mantisa por la base 10 elevada al exponente. En el caso de 1.004500e+03, tendríamos:

$$1.004500 \times 10^3 = 1.004500 \times 1000 = 1004.5$$

La ventaja de utilizar notación científica está en poder representar números muy grandes, o muy pequeños, en poco espacio. La diferencia entre los formatos **e** y **E** radica en cómo queramos exhibir la **e** que separa la mantisa del exponente; o bien una **e** minúscula o bien una **E** mayúscula. Los formatos **g** y **G** son llamados “genéricos” pues, dependiendo del número, son exhibidos como en el formato **f** o como en el **e** o el **E**. El tipo **%** simplemente multiplica el valor por 100 antes de imprimirlo; así, 0,05 es impreso como 5%. Vea algunos ejemplos en la lista 7.44.

► Lista 7.44 – Formateo de números decimales

```
>>> "{:8e}".format(3.141592653589793)
'3.141593e+00'
>>> "{:8E}".format(3.141592653589793)
'3.141593E+00'
>>> "{:8g}".format(3.141592653589793)
' 3.14159'
>>> "{:8G}".format(3.141592653589793)
```

```
' 3.14159'
>>> "{:8g}".format(3.14)
'    3.14'
>>> "{:8G}".format(3.14)
'    3.14'
>>> "{:5.2%}".format(0.05)
'5.00%'
```

7.10 Juego del ahorcado

Veamos un juego muy simple, pero que ayuda a trabajar con cadenas de caracteres. El juego del ahorcado es simple y puede ser divertido.

► Lista 7.45 – Juego del ahorcado

```
palabra = input("Digite la palabra secreta:").lower().strip() ❶
for x in range(100):
    print() ❷
    digitadas = []
    aciertos = []
    errores = 0
    while True:
        contraseña=""
        for letra in palabra:
            contraseña +=letra if letra in aciertos else "." ❸
        print(contraseña)
        if contraseña == palabra:
            print("¡usted acertó!")
            break
        intento = input("\nDigite una letra:").lower().strip()
        if intento in digitadas:
            print("¡usted ya intentó esta letra!")
            continue ❹
        else:
            digitadas += intento
            if intento in palabra:
                aciertos += intento
            else:
                errores += 1
                print("¡usted erró!")
    print("X==:==\nX : ")
```

```

print("X 0 " if errores >= 1 else "X")
línea2=""
if errores == 2:
    línea2 = " | "
elif errores == 3:
    línea2 = " \|"
elif errores >= 4:
    línea2 = " \|| "
print("X%s" % línea2)
línea3=""
if errores == 5:
    línea3+=" / "
elif errores>=6:
    línea3+=" / \ "
print("X%s" % línea3)
print("X\n=====")
if errores == 6:
    print("¡Ahorcado!")
    break

```

Vea como es simple escribir un juego del ahorcado en Python. Vamos analizar lo que hace esa lista. En ❶, aprovechamos el retorno de **input** para llamar los métodos de cadena de caracteres **lower** y **strip**. Observe como escribimos una llamada después de la otra. Esto es posible porque **input**, **lower** y **strip** retornan un objeto **cadena de caracteres**. El resultado final de la cadena de caracteres digitada por el usuario, convertida a letras minúsculas y con espacios en blanco eliminados en el comienzo y en el final, es atribuido a la variable **palabra**. Es esa variable la que va a almacenar la palabra a ser adivinada por el jugador.

En ❷ saltamos varias líneas para que el jugador no vea lo que fue digitado como **palabra**. La idea es que un jugador escriba una palabra secreta y que otro trate de descubrirla.

En ❸, tenemos una nueva forma de condición que facilita decisiones simples. El **if** inmediato, o en la misma línea, sirve para decidir el valor a retornar, dependiendo de una condición. Es como el **if** que ya conocemos, pero el valor verdadero queda a la izquierda del **if**, y la condición a su derecha. El **else** no tiene **:** e indica el resultado en caso que la condición sea falsa. En el ejemplo, letra es el valor a retornar si la condición **letra in aciertos** es verdadera, y “.” es el valor si el resultado es falso. Así:

```

contraseña += letra if letra in aciertos else "."

```

sustituye:

```

if letra in aciertos:
    contraseña += letra
else:

```


contraseña += "."

La ventaja de esa construcción es que escribimos todo en una sola línea. Ese recurso es interesante para expresiones simples y no debe ser utilizado para todo y cualquier caso. En el juego del ahorcado usamos esa construcción para mostrar la palabra secreta en la pantalla, pero sustituyendo todas las letras aún no adivinadas por un ".".

En ❹, utilizamos la instrucción **continue**. La instrucción **continue** es similar al **break** que ya conocemos, pero sirve para indicar que debemos ignorar todas las líneas hasta el fin de la repetición y volver al comienzo, sin terminarla. En el juego del ahorcado eso hace que la ejecución pase de ❹ para el **while** directamente, saltando todas las líneas después de ella. Cuando es utilizada con **while**, **continue** causa la reevaluación de la condición de la repetición; cuando es utilizada con **for**, hace que el próximo elemento sea utilizado. Considere cómo **continue** va hacia el fin de la repetición y vuelve a la línea de **for** o **while**.

Ejercicio 7.7 Modifique el programa de modo tal de escribir la palabra secreta en caso que el jugador pierda.

Ejercicio 7.8 Modifique el juego del ahorcado de modo de utilizar una lista de palabras. En el comienzo, pregunte un número y calcule el índice de la palabra a utilizar por la fórmula: $\text{índice} = (\text{número} * 776) \% \text{len}(\text{lista_de_palabras})$.

Ejercicio 7.9 Modifique el programa para utilizar listas de cadenas de caracteres para dibujar el muñeco del ahorcado. Puede utilizar una lista para cada línea y organizarlas en una lista de listas. En vez de controlar cuándo imprimir cada parte, dibuje en esas listas, sustituyendo el elemento a dibujar.

Ejemplo:

```
>>> línea = list("X-----")
>>> línea
['X', '-', '-', '-', '-', '-', '-']
>>> línea[6] = "|"
>>> línea
['X', '-', '-', '-', '-', '-', '|']
>>> "".join(línea)
'X-----|'
```

Ejercicio 7.10 Escriba un ta-te-ti (tres en raya) para dos jugadores. El juego debe preguntar dónde quiere jugar y alternar entre los jugadores. En cada jugada, verifique si la posición está libre. Verifique también cuándo un jugador ganó la partida. Un ta-te-ti puede ser visto como una lista de 3 elementos, donde cada elemento es otra lista, también con tres elementos.

Ejemplo del juego:

```
X | O |
---+---+---
| X | X
---+---+---
|   | O
```

Donde cada posición puede ser vista como un número. Verifique abajo un ejemplo de las posiciones mapeadas para la misma posición de su teclado numérico.

7		8		9
---+---+---				
4		5		6
---+---+---				
1		2		3

Funciones

En Python podemos definir nuestras propias funciones. Sabemos cómo usar varias funciones como `len`, `int`, `float`, `print` e `input`. En este capítulo veremos cómo declarar nuevas funciones y utilizarlas en programas.

Para definir una nueva función utilizaremos la instrucción `def`. Veamos cómo declarar una función de suma que recibe dos números como parámetros y los imprime en la pantalla (Lista 8.1).

►Lista 8.1 – Definición de una nueva función

```
def suma(a,b): ❶
    print(a+b) ❷
suma(2,9) ❸
suma(7,8)
suma(10,15)
```

Observe en ❶ que usamos la instrucción `def` seguida por el nombre de la función, en el caso, `suma`. Después del nombre y entre paréntesis, especificamos el nombre de los parámetros que la función recibirá. Llamamos al primero `a` y al segundo `b`. Observe también que usamos `:` después de los parámetros para indicar el comienzo de un bloque.

En ❷, usamos la función `print` para exhibir `a+b`. Observe que escribimos ❷ dentro del bloque de la función, o sea, más a la derecha.

A diferencia de lo que vimos hasta ahora, esas líneas no serán ejecutadas inmediatamente, excepto la definición de la función en sí. En realidad, la definición prepara al intérprete para ejecutar la función cuando ésta es llamada en otras partes del programa. Para llamar una función definida en el programa, haremos del mismo modo que las funciones ya definidas en el lenguaje: escribiremos el nombre de la función seguido de los parámetros entre paréntesis. Ejemplos de cómo llamar la función `suma` son presentados a partir de ❸. En el primer ejemplo, llamamos `suma(2,9)`. En ese caso, la función será llamada con `a` valiendo 2, y `b` valiendo 9. Los parámetros son sustituidos en el mismo orden en que fueron definidos, o sea, el primer valor como `a` y el segundo como `b`.

Las funciones son especialmente interesantes para aislar una tarea específica en un tramo de programa. Eso permite que la solución de un problema sea reutilizada en otras partes del programa, sin necesidad de repetir las mismas líneas. El ejemplo anterior utiliza dos parámetros e imprime su suma. Esa función no retorna valores como la función `len` o `int`. Vamos a reescribir esa función de modo que el valor de la suma sea retornado (Lista 8.2).

►Lista 8.2 – Definición del retorno de un valor

```
def suma(a,b):
    return(a+b) ❶
print(suma(2,9))
```

Vea que ahora utilizamos la instrucción **return** ❶ para indicar el valor a retornar. Observe también que retiramos el **print** de la función. Esto es interesante porque la suma y la impresión de la suma de dos números son dos problemas diferentes. No siempre vamos a sumar e imprimir la suma, por eso vamos dejar que la función realice solo el cálculo. Si necesitamos imprimir el resultado, podemos utilizar la función **print**, como en el ejemplo.

Veamos otro ejemplo, una función que retorne verdadero o falso, dependiendo si el número es par o impar (Lista 8.3).

►Lista 8.3 – Retornando verdadero o falso si el valor es par o no

```
def espar(x):
    return(x%2==0)
print(espar(2))
print(espar(3))
print(espar(10))
```

Imagine ahora que necesitamos definir una función para retornar la palabra par o impar. Podemos reutilizar **espar** en otra función (Lista 8.4).

►Lista 8.4 – Reutilización de la función espar en otra función

```
def espar(x):
    return(x%2==0)
def par_o_impar(x):
    if espar(x): ❶
        return "par" ❷
    else:
        return "impar" ❸
print(par_o_impar(4))
print(par_o_impar(5))
```

En ❶ llamamos la función **espar** dentro de la función **par_o_impar**. Observe que no hay nada de especial en esa llamada: solo pasamos el valor de **x** para la función **espar** y utilizamos su retorno en el **if**. Si la función retorna verdadero, retornaremos “par” en ❷; en caso contrario, “impar” en ❸. Utilizamos dos **return** en la función **par_o_impar**. La instrucción **return** hace que la función pare de ejecutarse y que el valor sea retornado inmediatamente al programa o función que la llamó. Así, podemos entender la instrucción **return** como una interrupción de la ejecución de la función, seguida del retorno del valor. Las líneas de la función después de la instrucción **return** son ignoradas de forma similar a la instrucción **break** dentro de un **while** o un **for**.

Ejercicio 8.1 Escriba una función que retorne el mayor de dos números.

Valores esperados:

máximo(5,6) == 6

máximo(2,1) == 2

máximo(7,7) == 7

Ejercicio 8.2 Escriba una función que reciba dos números y retorne **True** si el primer número es múltiplo del segundo.

Valores esperados:

múltiplo(8,4) == **True**

múltiplo(7,3) == **False**

múltiplo(5,5) == **True**

Ejercicio 8.3 Escriba una función que reciba el lado (l) de un cuadro y retorne su área (A = lado²).

Valores esperados:

área_cuadrado(4) == 16

área_cuadrado(9) == 81

Ejercicio 8.4 Escriba una función que reciba la base y la altura de un triángulo y retorne su área (A = (base x altura)/2).

Valores esperados:

área_triángulo(6, 9) == 27

área_triángulo(5, 8) == 20

Veamos ahora un ejemplo de función de investigación en una lista (Lista 8.5).

►Lista 8.5 – Investigación en una lista

```
def investigue(lista, valor):  
    for x,e in enumerate(lista):  
        if e == valor:  
            return x ❶  
    return None ❷
```

```
L=[10, 20, 25, 30]
```

```
print(investigue(L, 25))
```

```
print(investigue(L, 27))
```

La función **investigue** recibe dos parámetros: la **lista** y el **valor** a investigar. Si el **valor** es encontrado, retornaremos el valor de su posición en ❶. En caso que no sea encontrado, retornaremos **None** en ❷. Observe que si retornamos en ❶, tanto el **for** como el **return** en ❷ son completamente ignorados. Decimos que **return** marca el fin de la ejecución de la función. Usando estructuras de repetición y condicionales, podemos programar dónde y cómo las funciones

retornarán, así como decidir el valor a ser retornado.

Veamos otro ejemplo, calculando la media de los valores de una lista (Lista 8.6).

►Lista 8.6 – Cálculo de la media de una lista

```
def suma(L):  
    total=0  
    for e in L:  
        total+=e  
    return total  
  
def media(L):  
    return(suma(L)/len(L))
```

Para calcular la media, definimos otra función llamada **suma**. De esa forma tendremos dos funciones, una para calcular la media y otra para calcular la suma de la lista. Definir las funciones de esa forma es más interesante, pues una función debe resolver solo un problema. Podríamos también tener definida la función **media**, como en la lista 8.7.

►Lista 8.7 – Suma y cálculo de la media de una lista

```
def media(L):  
    total=0  
    for e in L:  
        total+=e  
    return total/len(L)
```

Cuál de las dos formas se debe elegir, depende del tipo de problema que se quiere resolver. Si no necesita el valor de la suma de los elementos de una lista en ninguna parte del programa, la segunda forma es interesante. La primera (suma y media definidas en dos funciones separadas) es más interesante a largo plazo y es también una buena práctica de programación. A medida que vamos creando funciones, podemos almacenarlas para usarlas en otros programas. Con el tiempo, usted va a coleccionar esas funciones, creando una biblioteca o módulo. Veremos más sobre eso cuando hablemos de importación y módulos. Por ahora, trate de fijar dos reglas: una función debe resolver solamente un problema y, cuanto más genérica sea su solución, mejor será a largo plazo.

Para saber si su función resuelve solo un problema, trate de definirla sin utilizar la conjunción “y”. Si su función hace eso y aquello, ya es una señal que realiza más de una tarea y que tal vez tenga que ser desmembrada en otras funciones. No se preocupe ahora por definir funciones perfectas, pues a medida que vaya ganando experiencia en la programación esos conceptos se volverán más claros.

Resolver los problemas de la manera más genérica posible es prepararse para reutilizar la función en otros programas. El hecho que la función solo pueda ser utilizada en el mismo programa que la definió no es un gran problema, pues a medida que los programas se vuelven más complejos, exigen soluciones propias y detalladas. Mientras tanto, si todas sus funciones sirven solo para un programa, considere eso como una señal de alerta.

Veamos qué no debemos hacer en la lista 8.8.

►Lista 8.8 – Cómo no escribir una función

```
def suma(L):  
    total=0  
    x = 0  
    while x<5:  
        total+=L[x]  
        x+=1  
    return total  
L=[1,7,2,9,15]  
print(suma(L)) ❶  
print(suma([7,9,12,3,100,20,4])) ❷
```

¿usted sabría decir por qué la función funcionó en ❶, pero devolvió un valor incorrecto en ❷?

La forma cómo definimos la función suma no es genérica, o mejor dicho, solo funciona en casos donde debemos sumar listas con cinco elementos. Es ante ese tipo de errores que usted debe estar atento. Si la función debe sumar todos los elementos de cualquier lista pasada como parámetro, debemos al menos presumir que podemos pasar listas con tamaños diferentes. Explicaremos más sobre eso cuando hablemos de pruebas.

Un problema clásico de programación es el cálculo del factorial. El factorial de un número es utilizado en estadística para calcular el número de combinaciones y permutaciones de conjuntos. Su cálculo es simple, por eso es muy utilizado como ejemplo en cursos de programación. Para calcular el factorial multiplicamos el número por todos los números que lo preceden hasta llegar a 1.

Por ejemplo:

- factorial de 3: $3 \times 2 \times 1 = 6$.
- factorial de 4: $4 \times 3 \times 2 \times 1 = 24$

Y así sucesivamente. Un caso especial es el factorial de 0 que es definido como 1. Veamos una función que calcule el factorial de un número en la lista 8.9.

►Lista 8.9 – Cálculo del factorial

```
def factorial(n):  
    fat = 1  
    while n>1:  
        fat*=n  
        n-=1  
    return fat
```

Si rastrea esa función, verá que calculamos el factorial multiplicando el valor de **n** por el valor anterior (**fat**) y que empezamos del mayor número hasta llegar en 1. Observe que la forma en que definimos la función satisface el caso especial del factorial de cero.

Podríamos también tener definida la función como en la lista 8.10.

►Lista 8.10 – Otra forma de calcular el factorial

```
def factorial(n):  
    fat=1  
    x=1  
    while x<=n:  
        fat*=x  
        x+=1  
    return fat
```

Existen varias formas de resolver el problema, y todas pueden ser correctas. Para decidir si nuestra implementación es correcta, tenemos que observar los valores retornados y compararlos con los valores de referencia.

Ejercicio 8.5 Reescriba la función de la lista 8.5 de modo de utilizar los métodos de investigación de lista, vistos en el capítulo 7.

Ejercicio 8.6 Reescriba el programa de la lista 8.8 de modo de utilizar **for** en vez de **while**.

Consejo: Python tiene funciones para calcular la suma, el máximo y el mínimo de una lista.

```
>>> L=[5,7,8]  
>>> sum(L)  
20  
>>> max(L)  
8  
>>> min(L)  
5
```

8.1 Variables locales y globales

Cuando usamos funciones, empezamos a trabajar con variables internas o locales y con variables externas o globales. La diferencia entre ellas es la visibilidad u objetivo.

Una variable local de una función existe solo dentro de ella, siendo normalmente inicializada en cada llamada. Así, no podemos acceder al valor de una variable local fuera de la función que la creó y, por eso, pasamos parámetros y retornamos valores en las funciones, de modo de posibilitar el cambio de datos en el programa.

Una variable global es definida fuera de una función; puede ser vista por todas las funciones del módulo y por todos los módulos que importan el módulo que la definió.

Veamos un ejemplo de los dos casos:

```
EMPRESA="Unidos Venceremos Ltda"  
def imprime_cabecal():
```



```
print(EMPRESA)
print("-" * len(EMPRESA))
```

La función `imprime_cabecal` no recibe parámetros ni retorna valores. Ella simplemente imprime el nombre de la empresa y trazos abajo de él. Observe que utilizamos la variable `EMPRESA` definida fuera de la función. En ese caso, `EMPRESA` es una variable global, pudiendo ser accesada en cualquier función.

Las variables globales deben ser utilizadas lo menos posible en sus programas, pues dificultan la lectura y violan el encapsulamiento de la función. La dificultad de la lectura está en buscar definición y contenidos fuera de la función en sí, que pueden cambiar entre diferentes llamadas. Además de eso, una variable global puede ser alterada por cualquier función, volviendo más trabajosa la tarea de saber quién altera su valor realmente. El encapsulamiento se ve comprometido porque la función depende de una variable externa; una variable que no ha sido declarada dentro de la función ni recibida como parámetro.

Aunque debemos utilizar variables globales con cuidado, eso no significa que ellas no tengan uso o que puedan simplemente ser clasificadas como mala práctica.

Un buen uso de variables globales es guardar valores constantes que deben ser accesibles a todas las funciones del programa, como, por ejemplo, el nombre de la empresa.

Las variables globales también son utilizadas en nuestro programa principal para inicializar el módulo con valores iniciales. Trate de utilizar variables globales solo para configuración y con valores constantes. Por ejemplo, si en la función `imprime_cabecal` el nombre de la empresa cambiase entre una llamada y otra, no debería ser almacenado en una variable global, sino pasado por parámetro.

Debido a la capacidad del lenguaje Python de declarar variables a medida que las necesitamos, debemos tener cuidado cuando alteramos una variable global dentro de una función. Veamos un ejemplo:

```
a=5 ❶
def cambia_e_imprime():
    a=7 ❷
    print("A dentro de la función: %d" % a)
print("a antes de cambiar: %d" % a) ❸
cambia_e_imprime() ❹
print("a después de cambiar: %d" % a) ❺
```

Ejecute el programa y analice su resultado. ¿usted sabe por qué el valor de `a` no cambió? Trate de responder esa pregunta antes de continuar leyendo.

En ❶, creamos la variable global `a`. En ❷, tenemos la variable local de la función, también llamada `a`, recibiendo 7. En ❸ y ❺, imprimimos el valor de la variable global `a` y, en ❹, el valor de la variable local `a`. Para la computadora, esas variables son completamente diferentes, aunque tengan el mismo nombre. En ❹, podemos acceder a su contenido porque realizamos la impresión dentro de la función. Del mismo modo, la variable `a` que imprimimos es la variable local, con valor 7. Esa variable local deja de existir al final de la función y eso explica porqué no alteramos la variable global `a`.

Si queremos modificar una variable global dentro de una función, debemos informar que estamos usando una variable global antes de inicializarla, en la primera línea de nuestra función. Esa definición es hecha con la instrucción **global**. Veamos el ejemplo modificado:

```
a=5
def cambia_e_imprime():
    global a ❶
    a=7 ❷
    print("A dentro de la función: %d" % a)
print("a antes de cambiar: %d" % a)
cambia_e_imprime()
print("a después de cambiar: %d" % a)
```

Ahora, en ❶, estamos trabajando con la variable **a** global. Así, cuando hicimos **a=7** en ❷, cambiamos el valor de **a**.

Recuerde limitar el uso de variables globales.

La utilización de la instrucción **global** puede señalar un problema de construcción en el programa. Recuerde utilizar variables globales solo para configuración y, de preferencia, como constantes. De esa forma, usted utilizará **global** solo en las funciones que alteran la configuración de su programa o módulo. La instrucción **global** facilita la elección de nombres de variables locales en los programas, pues deja claro que estamos usando una variable preexistente y no creando una nueva.

Trate de definir variables globales de modo de facilitar la lectura de su programa, como usar solo letras mayúsculas en sus nombres o un prefijo, por ejemplo, **EMPRESA** o **G_EMPRESA**.

8.2 Funciones recursivas

Una función puede llamarse a sí misma. Cuando eso ocurre tenemos una función recursiva. El problema del factorial es interesante para demostrar el concepto de recursividad. Podemos definir el factorial de un número como ese número multiplicado por el factorial de su antecesor. Si llamamos a nuestro número **n**, tenemos una definición recursiva del factorial como:

$$factorial(n) = \begin{cases} 1 & 0 \leq n \leq 1 \\ n \times factorial(n-1) & \end{cases}$$

En Python, definiríamos la función recursiva para cálculo del factorial como el programa de la lista 8.11.

►Lista 8.11 – Función recursiva del factorial

```
def factorial(n):
    if n==0 or n == 1:
        return 1
    else:
        return n*factorial(n-1)
```

Vamos a adicionar algunas líneas para facilitar el rastreo, alterando el programa para imprimir en la entrada de la función (Lista 8.12).

►Lista 8.12 – Función modificada para facilitar el rastreo

```
def factorial(n):
    print("Calculando el factorial de %d" % n)
    if n==0 or n == 1:
        print("Factorial de %d = 1" % n)
        return 1
    else:
        fat = n*factorial(n-1)
        print(" factorial de %d = %d" % (n, fat) )
    return fat
factorial(4)
```

Que produce como resultado la pantalla de la lista 8.13.

►Lista 8.13 – Cálculo del factorial de 4

```
Calculando el factorial de 4
Calculando el factorial de 3
Calculando el factorial de 2
Calculando el factorial de 1
Factorial de 1 = 1
factorial de 2 = 2
factorial de 3 = 6
factorial de 4 = 24
```

La secuencia de Fibonacci es otro problema clásico en el cual podemos aplicar funciones recursivas. La secuencia puede ser entendida como el número de parejas de conejos que tendríamos después de cada ciclo de reproducción, considerando que cada ciclo da origen a una pareja. Aunque matemáticamente la secuencia de Fibonacci revele propiedades más complejas, nos limitaremos a la historia de las parejas de conejos.

La secuencia comienza con dos números 0 y 1. Los números siguientes son la suma de los dos anteriores. La secuencia quedaría así: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Así, una función para calcular el enésimo término de la secuencia de Fibonacci puede ser definida como:

$$fibonacci(n) = \begin{cases} n & n \leq 1 \\ fibonacci(n-1) + fibonacci(n-2) & \end{cases}$$

Siendo implementada en Python como es presentado en la lista 8.14.

►Lista 8.14 – Función recursiva de Fibonacci

```
def fibonacci(n):
    if n<=1:
        return n
    else:
        return fibonacci(n-1)+fibonacci(n-2)
```

Ejercicio 8.7 Defina una función recursiva que calcule el mayor divisor común (M.D.C.) entre dos números a y b, donde $a > b$.

$$mdc(a, b) = \begin{cases} a & b = 0 \\ mdc(b, a - b \lfloor \frac{a}{b} \rfloor) & a > b \end{cases}$$

Donde $a - b \lfloor \frac{a}{b} \rfloor$ puede ser escrito en Python como: `a % b`.

Ejercicio 8.8 Usando la función mdc definida en el ejercicio anterior, defina una función para calcular el menor múltiplo común (M.M.C.) entre dos números.

$$mmc(a, b) = \frac{|a \times b|}{mdc(a, b)}$$

Donde $|a \times b|$ puede ser escrito en Python como: `abs(a*b)`.

Ejercicio 8.9 Rastree el programa de la lista 8.12 y compare el resultado con el presentado en la lista 8.13.

Ejercicio 8.10 Reescriba la función para cálculo de la secuencia de Fibonacci, sin utilizar recursión.

8.3 Validación

Las funciones son muy útiles para validar la entrada de datos. Veamos el código para leer un valor entero, limitado por un valor de mínimo y máximo (Lista 8.15). Ese tramo repite la entrada de datos hasta tener un valor válido.

►Lista 8.15 – Ejemplo de validación sin usar una función

```
while True:
    v=int(input("Digite un valor entre 0 y 5:"))
    if v<0 or v>5:
        print("Valor inválido.")
    else:
        break
```

Podemos transformarlo en una función que reciba la pregunta y los valores de máximo y mínimo (Lista 8.16).

►Lista 8.16 – Validación de entero usando función

```
def franja_int(pregunta, mínimo, máximo):
    while True:
        v=int(input(pregunta))
        if v<mínimo or v>máximo:
            print("Valor inválido. Digite un valor entre %d y %d" % (mínimo, máximo))
        else:
            return v
```

Ese tipo de verificación es muy importante cuando nuestro programa solo funciona con una franja de valores. Cuando verificamos los datos del programa, estamos realizando una validación. La validación es muy importante para evitar errores difíciles de detectar después de haber escrito el programa. Siempre que su programa reciba datos, sea por el teclado o por un archivo, verifique si esos datos están en la franja y en el formato adecuado para un buen funcionamiento.

Ejercicio 8.11 Escriba una función para validar una variable, cadena de caracteres. Esa función recibe como parámetro la cadena de caracteres, el número mínimo y máximo de caracteres. Retorne verdadero si el tamaño de la cadena de caracteres está entre los valores de máximo y mínimo, y falso en caso contrario.

Ejercicio 8.12 Escriba una función que reciba una cadena de caracteres y una lista. La función debe comparar la cadena de caracteres pasada con los elementos de la lista, también pasada como parámetro. Retorne verdadero si la cadena de caracteres es encontrada dentro de la lista, y falso en caso contrario.

8.4 Parámetros opcionales

No siempre necesitaremos pasar todos los parámetros para una función, prefiriendo utilizar un valor previamente elegido como estándar, pero dejando la posibilidad de alterarlo, en caso necesario. Veamos una función que imprime una barra en la pantalla en la lista 8.17.

►Lista 8.17 – Función para imprimir una barra en la pantalla

```
def barra():
    print("*" * 40)
```

En este ejemplo, la función **barra** no recibe ningún parámetro y puede ser llamada con **barra()**. Sin embargo, tanto el asterisco (*) como la cantidad de caracteres a exhibir en la barra pueden necesitar ser alterados. Por ejemplo, podemos utilizar una barra de asteriscos en una parte de nuestro programa y una barra con puntos en otra. Para resolver ese problema podemos utilizar parámetros opcionales.

►Lista 8.18 – Función para imprimir una barra en la pantalla con parámetros opcionales

```
def barra(n=40, carácter="*"):
    print(carácter * n)
```

Esa nueva definición (Lista 8.18) permite utilizar la función barra como antes, o sea, **barra()**, donde

carácter será igual a “*” y n será igual a 40. Al usar parámetros opcionales estamos especificando qué valores deben ser utilizados en caso que un nuevo valor no sea especificado, pero aún así permitiendo la posibilidad de pasar otro valor. Vea los ejemplos de la lista 8.19.

►Lista 8.19 – Pasaje de parámetros opcionales

```
>>> barra(10) # hace que n sea 10
*****
>>> barra(10,"-") # n = 10 y carácter="-"
-----
```

Los parámetros opcionales son útiles para evitar el pasaje innecesario de los mismos valores, pero preservando la opción de pasar valores, si necesario.

Podemos combinar parámetros opcionales y obligatorios en la misma función. Veamos la función `suma` en la lista 8.20.

►Lista 8.20 – Función `suma` con parámetros obligatorios y opcionales

```
def suma(a, b, imprime=False):
    s = a + b
    if imprime:
        print(s)
    return s
```

En el ejemplo, `a` y `b` son parámetros obligatorios e `imprime` es un parámetro opcional. Esa función puede ser utilizada como muestra la lista 8.21.

►Lista 8.21 – Uso de la función `suma` con parámetros obligatorios y opcionales

```
>>> suma(2,3)
5
>>> suma(3,4, True)
7
7
>>> suma(5,8, False)
13
```

Aunque podamos combinar parámetros opcionales y obligatorios, los mismos no pueden ser mezclados entre sí, y los parámetros opcionales siempre deben ser los últimos. Veamos una definición inválida en la lista 8.22.

►Lista 8.22 – Definición inválida de la función `suma` con parámetros opcionales antes de los obligatorios

```
def suma(imprime=True, a, b):
    s = a + b
    if imprime:
        print(s)
```

```
return s
```

Esa definición es inválida porque el parámetro opcional `imprime` es seguido por parámetros obligatorios `a` y `b`.

8.5 Nombrando parámetros

Python soporta la llamada de funciones con varios parámetros, pero hasta ahora vimos solo el caso en que hicimos la llamada de la función pasando los parámetros en el mismo orden en que fueron definidos. Cuando especificamos el nombre de los parámetros, podemos pasarlos en cualquier orden. Veamos el ejemplo de la función `rectángulo`, definida en la lista 8.23.

►Lista 8.23 – Función `rectángulo` con parámetros obligatorios y opcionales

```
def rectángulo(ancho, altura, carácter="*"):
    línea = carácter * ancho
    for i in range(altura):
        print(línea)
```

La función `rectángulo` puede ser llamada como ejemplifica la lista 8.24.

►Lista 8.24 – Llamando a la función `rectángulo`, nombrando los argumentos

```
>>> rectángulo(3,4)
***
***
***
***
>>> rectángulo(ancho=3, altura=4)
***
***
***
***
>>> rectángulo(altura=4, ancho=3)
***
***
***
***
>>> rectángulo(carácter="-", altura=4, ancho=3)
---
---
---
---
```

Una vez que utilizamos el nombre de cada parámetro para hacer la llamada, el orden de pasaje deja

de ser importante. Observe también que el parámetro **carácter** sigue siendo opcional, pero si lo especifica, también debemos hacerlo con nombre.

Cuando especificamos el nombre de un parámetro, estamos obligados a especificar el nombre de todos los otros parámetros también. Por ejemplo, las llamadas que siguen son inválidas:

►Lista 8.25 – Llamadas inválidas de la función **rectángulo**

```
rectángulo(ancho=3, 4)
rectángulo(ancho=3, altura=4, "*")
```

Una excepción a esa regla se da cuando combinamos parámetros obligatorios y opcionales. Por ejemplo, podemos pasar los parámetros obligatorios sin usar sus nombres, pero respetando el orden usado en la declaración, y emplear parámetros nombrados solo para elegir qué parámetros opcionales usar. Recuerde que al pasar el primer parámetro nombrado (usando su nombre), todos los parámetros siguientes también deben tener sus nombres especificados.

8.6 Funciones como parámetro

Un poderoso recurso de Python es permitir el pasaje de funciones como parámetro. Eso permite combinar varias funciones para realizar una tarea. Veamos un ejemplo en la lista 8.26.

►Lista 8.26 – Funciones como parámetro

```
def suma(a,b):
    return a+b
def sustracción(a,b):
    return a-b
def imprime(a,b, foper):
    print(foper(a,b)) ❶
imprime(5,4, suma) ❷
imprime(10,1, sustracción) ❸
```

Las funciones **suma** y **sustracción** fueron definidas normalmente, pero la función **imprime** recibe un parámetro llamado **foper**. En ese caso, **foper** es la función que pasaremos como parámetro. En ❶, pasamos los parámetros **a** y **b** para la función **foper** que aún no conocemos. Observe que pasamos **a** y **b** a **foper** como haríamos con cualquier otra función. En ❷, llamamos la función **imprime**, pasando la función **suma** como parámetro. Observe que, para pasar la función **suma** como parámetro, escribimos solo su nombre, sin pasar cualquier parámetro o usar paréntesis, como haríamos con una variable normal. En ❸, llamamos a la función **imprime**, pero esta vez pasando la función **sustracción** como **foper**.

Pasar funciones como parámetro permite inyectar funcionalidades dentro de otras funciones, volviéndolas configurables y más genéricas. Veamos otro ejemplo en la lista 8.27.

►Lista 8.27 – Configuración de funciones con funciones

```
def imprime_lista(L, fimpresión, fcondición):
```



```

for e in L:
    if fcondición(e):
        fimpresión(e)
def imprime_elemento(e):
    print("Valor: %d" % e)
def espar(x):
    return x % 2 == 0
def esimpar(x):
    return not espar(x)
L=[1,7,9,2,11,0]
imprime_lista(L, imprime_elemento, espar)
imprime_lista(L, imprime_elemento, esimpar)

```

La función `imprime_lista` recibe una lista y dos funciones como parámetro. La función `fimprime` es utilizada para realizar la impresión de cada elemento, pero solo es llamada si el resultado de la función pasada como `fcondición` es verdadero. En el ejemplo, llamamos a `imprime_lista` pasando la función `espar` como parámetro, haciendo que solo los elementos pares sean impresos. Después pasamos la función `esimpar`, de modo de imprimir solo elementos cuyo valor sea impar.

8.7 Empaquetamiento y desempaquetamiento de parámetros

Otra flexibilidad del lenguaje Python es poder pasar parámetros empaquetados en una lista. Veamos un ejemplo en la lista 8.28.

►Lista 8.28 – Empaquetamiento de parámetros en una lista

```

def suma(a,b):
    print(a+b)
L=[2,3]
suma(*L) ❶

```

En ❶, estamos utilizando el asterisco para indicar que queremos desempaquetar la lista `L` utilizando sus valores como parámetro para la función `suma`. En el ejemplo, `L[0]` será atribuido a `a` y `L[1]` a `b`. Ese recurso permite almacenar nuestros parámetros en listas y evita construcciones del tipo `suma(L[0], L[1])`.

Veamos otro ejemplo, donde utilizaremos una lista de listas para realizar varias llamadas a una función dentro de un **for** (Lista 8.29).

►Lista 8.29 – Otro ejemplo de empaquetamiento de parámetros en una lista

```

def barra(n=10, c="*"):
    print(c*n)
L=[[5, "-"], [10, "*"], [5], [6, "."]]
for e in L:

```

```
barra(*e)
```

Observe que aun usando el empaquetamiento de parámetros, recursos como parámetros opcionales aún son posibles cuando **e** contiene solo un elemento.

8.8 Desempaquetamiento de parámetros

Podemos crear funciones que reciben un número indeterminado de parámetros utilizando listas de parámetros (Lista 8.30).

►Lista 8.30 – Función suma con número indeterminado de parámetros

```
def suma(*args):  
    s=0  
    for x in args:  
        s+=x  
    return s
```

```
suma(1,2)
```

```
suma(2)
```

```
suma(5,6,7,8)
```

```
suma(9,10,20,30,40)
```

También podemos crear funciones que combinen parámetros obligatorios y una lista de parámetros (Lista 8.31).

►Lista 8.31 – Función imprime_mayor con número indeterminado de parámetros

```
def imprime_mayor(mensaje, *números):  
    mayor = None  
    for e in números:  
        if mayor == None or mayor < e:  
            mayor = e  
    print(mensaje, mayor)
```

```
imprime_mayor("Mayor:",5,4,3,1)
```

```
imprime_mayor("Max:", *[1,7,9])
```

Observe que el primer parámetro es **mensaje**, volviéndolo obligatorio. Así, **imprime_mayor()** retorna error, pues el parámetro **mensaje** no fue pasado, pero **imprime_mayor("Max:")** escribe **None**. Eso es porque **números** es recibido como una lista, pudiendo inclusive estar vacía.

8.9 Funciones Lambda

Podemos crear funciones simples, sin nombre, llamadas funciones lambda. Veamos un ejemplo en la lista 8.32.

►Lista 8.32 – Función lambda que recibe un valor y retorna el doble del mismo

```
a=lambda x: x*2 ❶
```

```
print(a(3)) ❷
```

En ❶, definimos una función lambda que recibe un parámetro, en el caso *x*, y que retorna el doble de ese número. Observe que todo fue hecho en una sola línea. La función es creada y atribuida a la variable *a*. En ❷, utilizamos *a* como una función normal.

Las funciones lambda también pueden recibir más de un parámetro, como muestra la lista 8.33.

►Lista 8.33 – Función lambda que recibe más de un parámetro

```
aumento=lambda a,b: (a*b/100)
```

```
aumento(100, 5)
```

8.10 Módulos

Después de crear varias funciones, los programas quedaron muy grandes. Necesitamos almacenar nuestras funciones en otros archivos y, de alguna forma, usarlas sin necesidad de reescribirlas, o peor aún, de copiarlas y pegarlas.

Python resuelve ese problema con módulos. Todo archivo *.py* es un módulo, pudiendo ser importado con el comando **import**.

Veamos dos programas: *entrada.py* (Lista 8.34) y *suma.py* (Lista 8.35).

►Lista 8.34 – Módulo entrada (entrada.py)

```
def valida_entero(mensaje, mínimo, máximo):  
    while True:  
        try:  
            v=int(input(mensaje))  
            if v >= mínimo and v <= máximo:  
                return v  
        else:  
            print("Digite un valor entre %d y %d" % (mínimo, máximo))  
    except:  
        print("usted debe digitar un número entero")
```

►Lista 8.35 – Módulo suma (suma.py) que importa entrada

```
import entrada  
L=[]  
for x in range(10):  
    L.append(entrada.valida_entero("Digite un número:", 0, 20))  
print("Suma: %d" % (sum(L)))
```

Utilizando el comando **import** fue posible llamar la función **valida_entero**, definida en *entrada.py*. Observe que para llamar la función **valida_entero** escribimos el nombre del módulo antes del

nombre de la función: `entrada.valida_entero`.

Usando módulos podemos organizar nuestras funciones en archivos diferentes y llamarlas cuando sea necesario, sin necesidad de reescribir todo.

No siempre queremos utilizar el nombre del módulo para acceder a una función: `valida_entero` puede ser más interesante que `entrada.valida_entero`.

Para importar la función `valida_entero` de modo de poder llamarla sin el prefijo del módulo, sustituya el **`import`** en la lista 8.35 por `from entrada import valida_entero`.

Después, modifique el programa sustituyendo `entrada.valida_entero` solo por `valida_entero`.

Debe utilizar ese recurso con atención, pues informar el nombre del módulo antes de la función es muy útil cuando los programas crecen, sirviendo de orientación para que se sepa qué módulo define tal función, facilitando su localización y evitando lo que se llama “conflicto de nombres”. Decimos que un conflicto de nombres ocurre cuando dos o más módulos definen funciones con nombres idénticos. En ese caso, utilizar la notación de llamada `módulo.función` resuelve el conflicto, pues la combinación del nombre del módulo con el nombre de la función es única, evitando la duda de descubrir el módulo que define la función que estamos llamando.

Otra construcción que debe ser utilizada con cuidado es `from entrada import *`. En este caso, estaríamos importando todas las definiciones del módulo `entrada`, y no solo la función `valida_entero`. Esa construcción es peligrosa porque si dos módulos definen funciones con el mismo nombre, la función utilizada en el programa será la del último **`import`**. Algo especialmente difícil de encontrar en programas grandes, y su uso no es aconsejable.

8.11 Números aleatorios

Una forma de generar valores para probar funciones y llenar listas es utilizar números aleatorios. Un número aleatorio puede ser entendido como un número obtenido al azar, sin ningún orden o secuencia predeterminada, como en un sorteo.

Para generar números aleatorios en Python, vamos a utilizar el módulo `random`. El módulo trae varias funciones para generación de números aleatorios y aun números generados con distribuciones no uniformes. Si quiere saber más sobre distribuciones, consulte un buen libro de estadística o Wikipedia (https://es.wikipedia.org/wiki/Distribuci%C3%B3n_de_probabilidad) para matar la curiosidad. Veamos la función `randint`, que recibe dos parámetros, siendo el primero el comienzo de la franja de valores a considerar para la generación de números; y el segundo, el final de esa franja. Tanto el comienzo como el final están incluidos en la franja.

►Lista 8.36 – Generando números aleatorios

```
import random
for x in range(10):
    print(random.randint(1,100))
```

En la lista 8.36, utilizamos `randint` con 1 y 100, luego, los números retornados deben estar en la franja entre 1 y 100. Si llamamos el número retornado de `x`, tendremos $1 \leq x \leq 100$. Cada vez que ejecutamos ese programa, se generarán valores diferentes. Esa diferencia también es interesante

si trabajamos con juegos o introducimos elementos de azar en nuestro programa.

► Lista 8.37 – Adivinando el número

```
import random
n=random.randint(1,10)
x=int(input("Elección un número entre 1 y 10:"))
if (x==n):
    print("¡usted acertó!")
else:
    print("¡usted erró!")
```

Vea el programa de la lista 8.37. Aun leyendo la lista, no podemos determinar la respuesta. En cada ejecución, un número diferente puede ser elegido. Ese tipo de aleatoriedad es utilizado en varios juegos y vuelve la experiencia única, haciendo que el mismo programa pueda ser utilizado varias veces con resultados diferentes.

Ejercicio 8.13 Altere el programa de la lista 8.37 de modo que el usuario tenga tres posibilidades de acertar el número. El programa termina si el usuario acierta o erra tres veces.

Podemos también generar números aleatorios fraccionarios o de punto flotante con la función `random` (Lista 8.38). La función `random` no recibe parámetros y retorna valores entre 0 y 1.

► Lista 8.38 – Números aleatorios entre 0 y 1 con `random`

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

Para obtener valores fraccionarios dentro de una determinada franja, podemos usar la función `uniform` (Lista 8.39).

► Lista 8.39 – Números aleatorios de punto flotante con `uniform`

```
import random
for x in range(10):
    print(random.uniform(15,25))
```

Podemos utilizar la función `sample` para elegir aleatoriamente elementos de una lista. Esa función recibe la lista y la cantidad de muestras (*samples*) o elementos que queremos retornar. Veamos cómo generar los números de un cartón del juego de a Lotto (Lista 8.40).

► Lista 8.40 – Selección de muestras de una lista aleatoriamente

```
import random
print(random.sample(range(1,101), 6))
```

Si queremos barajar los elementos de una lista, podemos utilizar la función `shuffle`. Ella recibe la lista a barajar, alterándola (Lista 8.41).

► Lista 8.41 – Acción de barajar elementos de una lista

```
import random
a=list(range(1,11))
random.shuffle(a)
print(a)
```

Ejercicio 8.14 Altere el programa de la lista 7.45, el juego del ahorcado. Elija la palabra para adivinar utilizando números aleatorios.

8.12 La función `type`

La función `type` retorna el tipo de una variable, función u objeto en Python. Veamos algunos ejemplos en el intérprete (Lista 8.42):

► Lista 8.42 – La función `type`

```
>>> a=5
>>> print(type(a))
<class 'int'>
>>> b="Hola"
>>> print(type(b))
<class 'str'>
>>> c=False
>>> print(type(c))
<class 'bool'>
>>> d=0.5
>>> print(type(d))
<class 'float'>
>>> f=print
>>> print(type(f))
<class 'builtin_function_or_method'>
>>> g=[]
>>> print(type(g))
<class 'list'>
>>> h={}
>>> print(type(h))
<class 'dict'>
>>> def función():
    pass
>>> print(type(función))
<class 'function'>
```

Observe que el tipo retornado es una clase. Veamos cómo utilizar esa información en un programa (Lista 8.43).

► Lista 8.43 – Utilizando la función `type` en un programa

```
import types

def dice_el_tipo(a):
    tipo = type(a)
    if tipo == str:
        return("Cadena de caracteres")
    elif tipo == list:
        return("Lista")
    elif tipo == dict:
        return("Diccionario")
    elif tipo == int:
        return("Número entero")
    elif tipo == float:
        return("Número decimal")
    elif tipo == types.FunctionType:
        return("Función")
    elif tipo == types.BuiltinFunctionType:
        return("Función interna")
    else:
        return(str(tipo))

print(dice_el_tipo(10))
print(dice_el_tipo(10.5))
print(dice_el_tipo("Hola"))
print(dice_el_tipo([1,2,3]))
print(dice_el_tipo({"a":1, "b":50}))
print(dice_el_tipo(print))
print(dice_el_tipo(None))
```

Vamos verificar cómo utilizar la función `type` para exhibir los elementos de una lista en la cual los elementos son de tipos diferentes. De esa forma puede ejecutar operaciones diferentes dentro de la lista, como verificar si un elemento es otra lista o un diccionario.

► Lista 8.44 – Usando `type` con los elementos de una lista

```
L=[ 2, "Hola", ["!"], {"a":1, "b":2}]

for e in L:
    print(type(e))
```

Veamos cómo navegar en una lista usando el tipo de sus elementos. Imagine una lista en la cual los

elementos pueden ser del tipo cadena de caracteres o lista. Si son del tipo cadena de caracteres, podemos simplemente imprimirlos. Si son del tipo lista, tendremos que imprimir cada elemento.

► Lista 8.45 – Navegando listas a partir del tipo de sus elementos

```
L=["a", ["b","c","d"], "e"]
```

```
for x in L:
```

```
    if type(x)==str:
```

```
        print(x)
```

```
    else:
```

```
        print("Lista:")
```

```
        for z in x:
```

```
            print(z)
```

Ejercicio 8.15 Utilizando la función `type` escriba una función recursiva que imprima los elementos de una lista. Cada elemento debe ser impreso separadamente, uno por línea. Considere el caso de listas dentro de listas, como `L=[1, [2,3,4,[5,6,7]]]`. En cada nivel, imprima la lista más a la derecha, como hacemos al identificar bloques en Python. Consejo: envíe el nivel actual como parámetro y utilícelo para calcular la cantidad de espacios en blanco a la izquierda de cada elemento.

Archivos

Necesitamos una forma de almacenar datos permanentemente. Los archivos son una excelente forma de entrada y salida de datos para programas. Con ellos podremos leer datos de otros programas, y aun de internet.

¿Pero, qué es un archivo? Un archivo es un área en el disco donde podemos leer y grabar información. Esa área es administrada por el sistema operativo de la computadora, por lo tanto no necesitamos preocuparnos sobre cómo está organizado el disco en ese espacio. Desde los programas podemos acceder a un archivo por su nombre y es el lugar donde podemos leer y escribir líneas de texto o datos en general.

Para acceder a un archivo necesitamos abrirlo. Durante la apertura, informamos el nombre del archivo, con el nombre del directorio donde se encuentra (si es necesario) y qué operaciones queremos realizar: lectura y/o escritura. En Python abrimos archivos con la función **open**.

La función **open** utiliza los parámetros nombre y modo. El nombre es el nombre del archivo en sí, por ejemplo, *leame.txt*. El modo indica las operaciones que vamos realizar (Tabla 9.1).

Tabla 9.1 – Modos de apertura de archivos

Modo	Operaciones
r	lectura (read)
w	escritura, borra el contenido si ya existe (write)
a	escritura, pero preserva el contenido si ya existe (append)
b	modo binario
+	actualización (lectura y escritura)

Los modos pueden ser combinados (“r+”, “w+”, “a+”, “r+b”, “w+b”, “a+b”). Las diferencias serán discutidas a continuación. La función **open** retorna un objeto del tipo **file**(archivo). Es ese objeto el que vamos a utilizar para leer y escribir los datos en el archivo. Utilizamos el método **write** para escribir o grabar datos en el archivo, **read** para leer y **close** para cerrarlo. Al trabajar con archivos, debemos siempre realizar el siguiente ciclo: apertura, lectura y/o escritura, cierre.

La apertura realiza la ligazón entre el programa y el espacio en el disco, administrado por el sistema operativo. Las etapas de lectura y/o escritura son las operaciones que deseamos realizar en el programa, y el cierre informa al sistema operativo que no vamos a trabajar más con el archivo.

El cierre del archivo es muy importante, pues cada archivo abierto consume recursos de la computadora. Sólo el cierre del archivo garantiza la liberación de esos recursos y preserva la integridad de los datos del archivo.

Veamos un ejemplo, donde vamos a escribir el archivo *números.txt* con 100 líneas. En cada línea, vamos a escribir un número (Lista 9.1).

►Lista 9.1 – Abriendo, escribiendo y cerrando un archivo

```
archivo=open("números.txt","w") ❶  
for línea in range(1,101): ❷  
    archivo.write("%d\n" % línea) ❸  
archivo.close() ❹
```

Ejecute el programa de la lista 9.1. Si todo corrió bien, nada aparecerá en la pantalla. El programa crea un nuevo archivo en el directorio actual, o sea, en el mismo directorio en que usted grabó su programa.

Si usa Windows, abra Windows Explorer, elija la carpeta donde graba sus programas y busque el archivo *números.txt*., puede abrirlo con el Notepad o simplemente haciendo clic dos veces en su nombre. Una vez abierto, observe las líneas: ¡su programa generó un archivo que puede ser abierto por otros programas!

Si utiliza el Mac OS X, active el Finder para localizar y abrir su archivo. En el Linux, puede utilizar el Nautilus o simplemente digitar `less números.txt` en la línea de comando.

En ❶, utilizamos la función `open` para abrir el archivo *números.txt*. Observe que el nombre del archivo es una cadena de caracteres y debe ser escrito entre comillas. El modo elegido fue "w", indicando escritura o grabación. El modo "w" crea el archivo si no existe. En caso que ya exista, su contenido es borrado. Para verificar ese efecto, ejecute el programa nuevamente y observe que el archivo continúa con 100 líneas. Pruebe también escribir algo en el archivo usando el Notepad (u otro editor de textos simple), grabe y ejecute el programa nuevamente. Todo lo que escribió en el editor se perdió, pues el programa borra el contenido del archivo y comienza todo de nuevo.

Creamos un `for` ❷ para generar los números de las líneas. En ❸, escribimos el número de la línea en el archivo, usando el método `write`. Observe que escribimos `archivo.write`, pues `write` es un método del objeto archivo. Observe que escribimos `"%d\n" % línea`, donde el `"%d"` funciona como en la función `print`, pero tenemos que adicionar el `"\n"` para indicar que queremos pasar a una nueva línea.

La línea ❹ cierra el archivo. El cierre garantiza que el sistema operativo fue informado que no vamos a trabajar más con el archivo. Esa comunicación es importante, pues el sistema operativo realiza diversas funciones para optimizar la velocidad de sus operaciones, y una de ellas es la de no grabar los datos directamente en el disco, sino en una memoria auxiliar. Cuando no cerramos el archivo, corremos el riesgo que esa memoria auxiliar no sea transferida al disco y, así, perder lo que fue escrito en el programa.

Veamos ahora un programa para leer el archivo e imprimir sus líneas en la pantalla (Lista 9.2).

►Lista 9.2 – Abriendo, leyendo y cerrando un archivo

```
archivo=open("números.txt","r") ❶  
for línea in archivo.readlines(): ❷  
    print(línea) ❸
```

```
archivo.close() ❹
```

En ❶, utilizamos la función `open` para abrir el archivo. El nombre es el mismo utilizado durante la grabación, pero el modo ahora es "r", o sea, lectura. En ❷ utilizamos el método `readlines`, que genera una lista en que cada elemento es una línea del archivo. En ❸ simplemente imprimimos la línea en la pantalla. En ❹ cerramos el archivo.

Vea que ❶ es muy parecida a la misma línea del programa anterior, y que ❹ es idéntica. Eso sucede porque la apertura y el cierre son partes esenciales en la manipulación de archivos. Siempre que manipulamos archivos, tendremos una apertura, operaciones y cierre.

Hasta ahora estamos trabajando con archivos del tipo texto. La característica principal de esos archivos es que su contenido es solo texto simple, con algunos caracteres especiales de control. El carácter de control más importante en un archivo texto es el que marca el fin de una línea. En Windows, el fin de línea es marcado por una secuencia de dos caracteres, cuyas posiciones en la tabla ASCII son 10 (LF – *Line Feed*, avance de línea) y 13 (CR – *Carriage Return*, retorno de carro). En el Linux, tenemos solo un carácter marcando el fin de línea, el *LineFeed* (10). En el Mac OS X, tenemos solo el *Carriage Return* (13) marcando el fin de una línea.

9.1 Parámetros de la línea de comando

Podemos acceder a los parámetros pasados al programa en la línea de comando utilizando el módulo `sys` y trabajando con la lista `argv`. Veamos el programa de la lista 9.3.

►Lista 9.3 – Impresión de los parámetros pasados en la línea de comando (fparam.py)

```
import sys
print("Número de parámetros: %d" % len(sys.argv))
for n,p in enumerate(sys.argv):
    print("Parámetro %d = %s" % (n,p))
```

Pruebe llamar al script en la línea de comando usando los siguientes parámetros:

```
fparam.py primero segundo tercero
fparam.py 1 2 3
fparam.py readme.txt 5
```

Observe que cada parámetro fue pasado con un elemento de la lista `sys.argv` y que los parámetros están separados por espacios en blanco en la línea de comando, pero que esos espacios son removidos en `sys.argv`. Si necesita pasar un parámetro con espacios en blanco, como un nombre formado por varias palabras, escríbalo entre comillas para que sea considerado como un único parámetro.

```
fparam.py "Nombre grande" "Segundo nombre grande"
```

Ejercicio 9.1 Escriba un programa que reciba el nombre de un archivo por la línea de comando y que imprima todas las líneas de ese archivo.
--

Ejercicio 9.2 Modifique el programa del ejercicio 9.1 para que reciba dos parámetros más: la línea de comienzo y la de fin para impresión. El programa debe imprimir solo las líneas entre esos dos valores (incluyendo las líneas de comienzo y fin).

9.2 Generación de archivos

Veamos el programa de la lista 9.4, que genera dos archivos de 500 líneas cada uno. El programa distribuye los números impares y pares en archivos diferentes.

Observe que para grabar en archivos diferentes, utilizamos un `if` y dos archivos abiertos para escritura. Utilizando el método `write` de los objetos `pares` e `impares`, hicimos la selección de dónde grabar el número. Observe que incluimos el `\n` para indicar fin de línea.

►Lista 9.4 – Grabación de números pares e impares en archivos diferentes

```
impares=open("impares.txt","w")
pares=open("pares.txt","w")
for n in range(0,1000):
    if n % 2 == 0:
        pares.write("%d\n" % n)
    else:
        impares.write("%d\n" % n)
impares.close()
pares.close()
```

9.3 Lectura y escritura

Podemos realizar diversas operaciones con archivos; entre ellas, leer: procesar y generar nuevos archivos. Utilizando el archivo *pares.txt* creado por el programa de la lista 9.4, veamos cómo filtrarlo de modo de generar un nuevo archivo, solo con números múltiplos de 4 (Lista 9.5).

►Lista 9.5 – Filtrado exclusivo de los múltiplos de cuatro

```
multiplos4=open("múltiplos de 4.txt","w")
pares=open("pares.txt")
for l in pares.readlines():
    if int(l) % 4 == 0: ❶
        multiplos4.write(l)
pares.close()
multiplos4.close()
```

Vea que en ❶ convertimos la línea leída de cadena de caracteres en un entero antes de hacer los cálculos.

Ejercicio 9.3 Cree un programa que lea los archivos *pares.txt* e *impares.txt* y que cree un único archivo *pareseimpares.txt* con todas las líneas de los otros dos archivos, de modo de preservar el orden numérico.

Ejercicio 9.4 Cree un programa que reciba el nombre de dos archivos como parámetros de la línea de comando y que genere un archivo de salida con las líneas del primero y del segundo archivo.

Ejercicio 9.5 Cree un programa que invierta el orden de las líneas del archivo *pares.txt*. La primera línea debe contener el mayor número; y la última, el menor.

9.4 Procesamiento de un archivo

Podemos también procesar las líneas de un archivo de entrada como si fuesen comandos. Veamos un ejemplo en la lista 9.6, en la cual las líneas cuyo primer carácter es igual a ";" serán ignoradas; las líneas con ">" serán impresas alineadas a la derecha; las líneas con "<", alineadas a la izquierda; y las que contengan "*" serán centradas.

Cree un archivo con las siguientes líneas y guárdelo como *entrada.txt*:

```
;Esta línea no debe ser impresa.  
>Esta línea debe ser impresa alineada a la derecha  
*Esta línea debe ser centrada  
Una línea normal  
Otra línea normal
```

►Lista 9.6 – Procesamiento de un archivo

```
ANCHO=79  
entrada=open("entrada.txt")  
for línea in entrada.readlines():  
    if línea[0]==";":  
        continue  
    elif línea[0]==">":  
        print(línea[1:].rjust(ANCHO))  
    elif línea[0]=="*":  
        print(línea[1:].center(ANCHO))  
    else:  
        print(línea)  
entrada.close()
```

Ejercicio 9.6 Modifique el programa de la lista 9.6 para imprimir 40 veces el símbolo "=" si este es el primer carácter de la línea. Adicione también la opción para parar de imprimir hasta que se presione la tecla enter cada vez que una línea comience con "." como primer carácter.

Ejercicio 9.7 Cree un programa que lea un archivo de texto y genere un archivo de salida paginado. Cada línea no debe contener más de 76 caracteres. Cada página tendrá como máximo 60 líneas. Adicione en la última línea de cada página el número de la página actual y el nombre del archivo original.

Ejercicio 9.8 Modifique el programa anterior para recibir también el número de caracteres por línea y el número de páginas por hoja por la línea de comando.

Ejercicio 9.9 Cree un programa que reciba una lista de nombres de archivo y los imprima, uno por uno.

Ejercicio 9.10 Cree un programa que reciba una lista de nombres de archivo y que genere solo un gran archivo de salida.

Ejercicio 9.11 Cree un programa que lea un archivo y cree un diccionario donde cada clave sea una palabra y cada valor sea el número de ocurrencias en el archivo.

Ejercicio 9.12 Modifique el programa anterior para registrar también la línea y la columna de cada ocurrencia de la palabra en el archivo. Para eso utilice listas en los valores de cada palabra, guardando la línea y la columna de cada ocurrencia.

Ejercicio 9.13 Cree un programa que imprima las líneas de un archivo. Ese programa debe recibir tres parámetros por la línea de comando: el nombre del archivo, la línea inicial y la última línea a imprimir.

Ejercicio 9.14 Cree un programa que lea un archivo de texto y elimine los espacios repetidos entre las palabras y al final de las líneas. El archivo de salida no debe tener más de una línea en blanco repetida.

Ejercicio 9.15 Altere el programa de la lista 7.5, el juego del ahorcado. Utilice un archivo en que una palabra sea grabada en cada línea. Use un editor de textos para generar el archivo. Al comenzar el programa, utilice ese archivo para cargar la lista de palabras. Pruebe también preguntar el nombre del jugador y generar un archivo con el número de aciertos de los cinco mejores.

Usando archivos podemos grabar datos de modo de reutilizarlos en los programas. Hasta ahora, todo lo que introdujimos o digitamos en los programas se perdía al final de la ejecución. Usando archivos, podemos registrar esa información y reutilizarla. Los archivos pueden ser utilizados para proveer una gran cantidad de datos a los programas. Veamos un ejemplo en el cual grabaremos nombres y teléfonos en un archivo de texto. Utilizaremos un menú para dejar que el usuario decida cuándo leer el archivo y cuándo grabarlo.

Ejecute el programa de la lista 9.7 y analícelo cuidadosamente.

►Lista 9.7 – Control de una agenda de teléfonos

```
agenda = []  
  
def pide_nombre():  
    return(input("Nombre: "))  
  
def pide_teléfono():  
    return(input("Teléfono: "))
```

```

def muestra_datos(nombre, teléfono):
    print("Nombre: %s Teléfono: %s" % (nombre, teléfono))

def pide_nombre_archivo():
    return(input("Nombre del archivo: "))

def investigación(nombre):
    mnombre = nombre.lower()
    for p, e in enumerate(agenda):
        if e[0].lower() == mnombre:
            return p
    return None

def nuevo():
    global agenda
    nombre = pide_nombre()
    teléfono = pide_teléfono()
    agenda.append([nombre, teléfono])

def borra():
    global agenda
    nombre = pide_nombre()
    p = investigación(nombre)
    if p!=None:
        del agenda[p]
    else:
        print("Nombre no encontrado.")

def altera():
    p = investigación(pide_nombre())
    if p!=None:
        nombre = agenda[p][0]
        teléfono = agenda[p][1]
        print("Encontrado:")
        muestra_datos(nombre, teléfono)
        nombre = pide_nombre()
        teléfono = pide_teléfono()
        agenda[p]=[nombre, teléfono]
    else:
        print("Nombre no encontrado.")

def lista():
    print("\nAgenda\n\n-----")
    for e in agenda:
        muestra_datos(e[0], e[1])

```

```

    print("-----\n")
def lee():
    global agenda
    nombre_archivo = pide_nombre_archivo()
    archivo = open(nombre_archivo, "r", encoding="utf-8")
    agenda = []
    for l in archivo.readlines():
        nombre, teléfono = l.strip().split("#")
        agenda.append([nombre, teléfono])
    archivo.close()
def graba():
    nombre_archivo = pide_nombre_archivo()
    archivo = open(nombre_archivo, "w", encoding="utf-8")
    for e in agenda:
        archivo.write("%s#%s\n" % (e[0], e[1]))
    archivo.close()
def valida_franja_entero(pregunta, inicio, fin):
    while True:
        try:
            valor = int(input(pregunta))
            if inicio <= valor <= fin:
                return(valor)
        except ValueError:
            print("Valor inválido, por favor digitar entre %d y %d" % (inicio, fin))
def menú():
    print("""
1 - Nuevo
2 - Altera
3 - Borra
4 - Lista
5 - Graba
6 - Lee
0 - Salir
""")
    return valida_franja_entero("Elección una opción: ",0,6)
while True:
    opción = menú()
    if opción == 0:

```



```

break
elif opción == 1:
    nuevo()
elif opción == 2:
    altera()
elif opción == 3:
    borra()
elif opción == 4:
    lista()
elif opción == 5:
    graba()
elif opción == 6:
    lee()

```

Los ejercicios a continuación modifican el programa de la lista 9.7.

Ejercicio 9.16 Explique cómo son almacenados los campos nombre y teléfono en el archivo de salida.

Ejercicio 9.17 Altere el programa para exhibir el tamaño de la agenda en el menú principal.

Ejercicio 9.18 ¿Qué sucede si el nombre o el teléfono contienen el carácter usado como separador en sus contenidos? Explique el problema y proponga una solución.

Ejercicio 9.19 Altere la función lista para que exhiba también la posición de cada elemento.

Ejercicio 9.20 Adicione la opción de ordenar la lista por nombre en el menú principal.

Ejercicio 9.21 En las funciones de **altera** y **borra**, pida que el usuario confirme la alteración y exclusión del nombre antes de realizar la operación en sí.

Ejercicio 9.22 Al leer o grabar una nueva lista, verifique si la agenda actual ya fue grabada. Puede usar una variable para controlar cuando la lista fue alterada (nuevo, altera, borra) y reinicializar ese valor cuando es leída o grabada.

Ejercicio 9.23 Altere el programa para leer la última agenda leída o grabada al inicializar. Consejo: utilice otro archivo para almacenar el nombre.

Ejercicio 9.24 ¿Qué sucede con la agenda si ocurre un error de lectura o grabación? Explíquelo.

Ejercicio 9.25 Altere las funciones `pide_nombre` y `pide_teléfono` de modo de recibir un parámetro opcional. En caso que ese parámetro sea pasado, utilícelo como retorno en caso que la entrada de datos esté vacía.

Ejercicio 9.26 Altere el programa de modo de verificar la repetición de nombres. Genere un mensaje de error en caso que dos entradas en la agenda tengan el mismo nombre.

Ejercicio 9.27 Modifique el programa para controlar también la comparación de cumpleaños y el e-mail de cada persona.

Ejercicio 9.28 Modifique el programa de modo de poder registrar varios teléfonos para la misma persona. Permita también registrar el tipo de teléfono: celular, fijo, residencia, trabajo o fax.

9.5 Generación de HTML

Las páginas web son la combinación de textos e imágenes interconectadas por links. Hoy, esa definición va más allá con la creciente popularidad del vídeo y las llamadas Aplicaciones Ricas de Internet (*Rich Internet Applications*), normalmente escritas en Javascript. Como aquí tratamos solo de introducir conceptos de programación, veamos cómo utilizar lo que ya sabemos sobre archivos para generar *home pages* o páginas web simples.

Toda página web está escrita en un lenguaje de marcación llamado HTML (*Hypertext Mark-up Language* – Lenguaje de Marcación de Hipertexto). Primero, necesitamos entender qué son “marcaciones”. Como el formato HTML es definido solo con texto simple (utiliza marcaciones), o sea, sin caracteres especiales de control. Estas son secuencias especiales de texto, delimitado por los caracteres de menor (<) y mayor (>). Esas secuencias son llamadas *tags* y pueden comenzar o finalizar un elemento. El elemento de más alto nivel de un documento HTML es llamado `<html>`. Escribiremos nuestras páginas web entre las tags `<html>` y `</html>`, donde la primera marca el comienzo del documento y la segunda su fin. Diremos que `<html>` es la *tag* que empieza el elemento *html*, y que `</html>` es la *tag* que lo finaliza. Vamos a seguir los consejos definidos en la especificación del estándar HTML 5 y definir la página inicial como en la lista 9.7. Observe que a la izquierda de cada línea presentamos su número. No debe digitar ese número en su archivo.

►Lista 9.7 – Página web simple hola.html

```
1  <!DOCTYPE html>
2  <html lang="es_ES">
3  <head>
4  <meta charset="utf-8">
5  <title>Título de la página</title>
6  </head>
7  <body>
8  ¡Hola!
9  </body>
```

10 </html>

Puede escribir esa página usando un editor de textos. Grabe el archivo con el nombre de **hola.html**. Ábrala en su browser (navegador) de internet preferido: Internet Explorer, FireFox, Chrome, Safari, Opera etc. La forma más fácil de abrir un archivo del disco en el browser es seleccionar el menú **Archivo** y después la opción **Abrir**. Elija el archivo *hola.html* y vea qué aparece en la pantalla. Si los acentos no aparecen correctamente, verifique si grabó el archivo como texto UTF-8, del mismo modo que hacemos con los programas.

Vamos a analizar esa página línea por línea. La línea 1 hace parte del formato y debe siempre ser incluida: ella indica que el documento fue escrito en el formato HTML.

La línea 2 contiene a *tag* **html**, pero no solo el nombre de la *tag* y el atributo **lang**, cuyo valor es **es_ES**. Observe que el comienzo de la *tag* es marcado con **<** y el fin con **>** y que **html lang="es_ES"** fue escrito en la misma *tag*, o sea, entre los símbolos de **<** y **>**. El atributo **lang** indica la lengua utilizada al escribir el documento: el valor **es_ES** indica español.

La línea 4 especifica que el archivo utiliza el formato de codificación de caracteres UTF-8 y permite escribir en español sin problemas. Mientras tanto, debe garantizar que realmente escribió el archivo usando UTF-8. En Windows, puede usar el editor PS-Pad para generar archivos UTF-8; en el Linux y en el Mac OS X ese tipo de codificación es usado por defecto (de manera predeterminada) y no debería causar problemas. Una señal de problema de codificación es si no consigue leer “Hola” en la pantalla o si el título de la página no aparece correctamente. Si altera el archivo HTML, usted debe solicitar que el browser lo recargue (*reload*) para visualizar los cambios.

La línea 5 trae el título de la página. Observe que el título está escrito entre **<title>** y **</title>**. Observe también que escribimos **title** y **meta** dentro del elemento **head** que comienza en la línea 3 y termina en la línea 6. El formato HTML especifica qué elementos deben ser escritos y dónde podemos escribirlos. Para más detalles puede consultar la especificación del formato HTML 5 en internet. Por ahora, considere que nuestro primer archivo html será utilizado como un modelo en nuestros programas.

Entre las líneas 7 y 9, definimos el elemento **body**, que es el cuerpo de la página web. La línea 8 contiene solo el mensaje **¡Hola!** Finalmente, en la línea 10 tenemos la tag que finaliza el elemento **html**, o sea, **</html>**, marcando el fin del documento. Pruebe alterar ese archivo, escribiendo un nuevo mensaje en la línea 8. Pruebe también digitar varias líneas de texto (Puede copiar y pegar un texto de 10 o 12 líneas). No se olvide de guardar el archivo con las modificaciones y de recargar la página en el browser para visualizarlas.

Como las páginas web son archivos de texto, podemos crearlas fácilmente en Python. Vea el programa de la lista 9.8 que crea una página HTML.

►Lista 9.8 – Creación de una página inicial en Python

```
página=open("página.html","w", encoding="utf-8")
página.write("<!DOCTYPE html>\n")
página.write("<html lang=\"es_ES\">\n")
página.write("<head>\n")
página.write("<meta charset=\"utf-8\">\n")
```

```

página.write("<title>Título de la página</title>\n")
página.write("</head>\n")
página.write("<body>\n")
página.write(";Hola!")
for l in range(10):
    página.write("<p>%d</p>\n" % l)
página.write("</body>\n")
página.write("</html>\n")
página.close()

```

Simplemente escribimos nuestra página dentro de un programa. Observe que al escribir comillas dentro de comillas, como en la tag `html`, donde el valor de `lang` es `utf-8` y está escrito entre comillas, tuvimos el cuidado de colocar una barra antes de las comillas, informando, de esa forma, que las comillas no hacen parte del programa y que no marcan el fin de la cadena de caracteres. Un detalle muy importante es el parámetro extra que pasamos en `open`. El parámetro `encoding="utf-8"` informa que queremos los archivos con la codificación UTF-8. Esa codificación tiene que ser la misma declarada en el archivo `html`, en caso contrario, tendremos problemas con caracteres acentuados.

Ejecute el programa y abra el archivo *pagina.html* en su browser. Modifique el programa para generar 100 párrafos en vez de 10. Ejecútelo nuevamente y vea el resultado en el browser (abriendo el archivo nuevamente o haciendo clic en recargar).

En nuestras primeras pruebas vimos que escribiendo varias líneas de texto en el cuerpo de la página no alteramos el formato de salida de la página. Eso es porque el formato `html` ignora espacios en blanco repetidos y quiebres de línea. Si queremos mostrar el texto en varios párrafos, debemos utilizar el elemento `p`. Así, las tags `<p>` y `</p>` marcan el comienzo y el fin de un párrafo, como en el programa de la lista 9.8.

Python ofrece recursos más interesantes para trabajar con cadenas de caracteres, como comillas triples que permiten escribir largos textos más fácilmente. Funcionan como las comillas, pero permiten digitar el mensaje en varias líneas. Veamos el programa de la lista 9.9.

►Lista 9.9 – Uso de comillas triples para escribir las cadenas de caracteres

```

página=open("página.html","w", encoding="utf-8")
página.write("""
<!DOCTYPE html>
<html lang="es_ES">
<head>
<meta charset="utf-8">
<title>Título de la página</title>
</head>
<body>
Hola!

```

```

"""
for l in range(10):
    página.write("<p>%d</p>\n" % l)
página.write("""
</body>
</html>
""")
página.close()

```

Otra ventaja es que no necesitamos colocar una barra antes de las comillas, pues ahora el fin de las comillas también es triple ("""), no siendo ya necesaria ninguna diferenciación. Pruebe ahora modificar el programa retirando los comandos `\n` del final de las líneas. Ejecútelo y vea el resultado en el browser. Debe percibir que aún sin utilizar quiebres de líneas en el archivo, el resultado permaneció igual. Eso es porque los espacios adicionales (repetidos) y los quiebres de línea son ignorados (o casi ignorados) en HTML. Mientras tanto, observe que la página creada es más difícil de leer para nosotros.

El formato HTML contiene innumerables tags que son continuamente revisadas y expandidas. Además de las tags que ya conocemos, tenemos también las que marcan los cabezales de los documentos: **h1**, **h2**, **h3**, **h4**, **h5** y **h6**. Los números de 1 a 6 son utilizados para indicar el nivel de la sección o subsección del documento, como haríamos en el Microsoft Word o en el OpenOffice, con los estilos de cabezales.

Veamos como generar páginas web a partir de un diccionario. Vamos a utilizar las claves como título de las secciones, y el valor como contenido del párrafo (Lista 9.10).

►Lista 9.10 – Generación de una página web a partir de un diccionario

```

películas={
    "drama": ["El ciudadano","El padrino"],
    "comedia": ["Tiempos Modernos","American Pie","Dr. Dolittle"],
    "policial": ["Lluvia Negra","El vengador anónimo","Duro de Matar"],
    "guerra": ["Rambo","Pelotón","Tora!Tora!Tora!"]
}

página=open("filmes.html","w", encoding="utf-8")
página.write("""
<!DOCTYPE html>
<html lang="es_ES">
<head>
<meta charset="utf-8">
<title>Filmes</title>
</head>
<body>
""")

```

```

for c, v in películas.items():
    página.write("<h1>%s</h1>" % c)
    for e in v:
        página.write("<h2>%s</h2>" % e)
página.write("""
</body>
</html>
""")
página.close()

```

Ejercicio 9.29 Modifique el programa de la lista 9.10 para utilizar el elemento `p` en vez de `h2` en las películas.

Ejercicio 9.30 Modifique el programa de la lista 9.10 para generar una lista html, usando los elementos `ul` y `li`. Todos los elementos de la lista deben estar dentro del elemento `ul`, y cada ítem dentro de un elemento `li`. Ejemplo:

```
<ul><li>Item1</li><li>Item2</li><li>Item3</li></ul>
```

Puede leer más sobre HTML y CSS (*Cascading Style-Sheets*) para generar páginas profesionales en Python.

9.6 Archivos y directorios

Ahora que ya sabemos leer, crear y alterar archivos, vamos aprender cómo listarlos, manipular directorios, verificar el tamaño y la comparación de creación de archivos en disco.

Para empezar, necesitamos que los programas sepan de dónde están siendo ejecutados. Vamos a utilizar la función `getcwd` del módulo `os` para obtener ese valor (Lista 9.11).

►Lista 9.11 – Obtención del directorio actual

```

>>> import os
>>> os.getcwd()

```

También podemos cambiar de directorio en Python, pero antes necesitamos crear algunos directorios para test. En la línea de comando, digite:

```

mkdir a
mkdir b
mkdir c

```

Si lo prefiere, puede crear las carpetas `a`, `b` y `c` usando Windows Explorer, el Finder u otro utilitario de su preferencia. Ahora, veamos la lista 9.12.

►Lista 9.12 – Cambio de directorio

```

import os
os.chdir("a")

```

```
print(os.getcwd())
os.chdir("..")
print(os.getcwd())
os.chdir("b")
print(os.getcwd())
os.chdir("../c")
print(os.getcwd())
```

La función `chdir` cambia el directorio actual, por eso la función `getcwd` presenta valores diferentes. Puede referenciar un archivo simplemente usando su nombre si el mismo está en el directorio actual de trabajo. Lo que hace `chdir` es cambiar el directorio de trabajo, permitiendo que acceda a sus archivos más fácilmente.

Cuando pasamos `".."` para `chdir` o cualquier otra función que manipule archivos y directorios, estamos refiriéndonos al directorio padre o de nivel superior. Por ejemplo, considere el directorio `z`, que contiene los directorios `h`, `i`, `j`. Dentro del directorio (o carpeta) `j`, tenemos el directorio `k` (Figura 9.1). Cuando un directorio está dentro de otro, decimos que el directorio padre contiene el directorio hijo. En el ejemplo, `k` es hijo de `j`, y `j` es padre de `k`. Tenemos también que `h`, `i`, `j` son todos hijos de `z`, o sea, `z` es el padre de `h`, `i`, `j`.



Figura 9.1– Estructura de directorios en Windows Explorer de Windows 8.1.

Ese tipo de direccionamiento es siempre relativo al directorio corriente (que está activo o en uso), por eso es tan importante saber en qué directorio estamos. Si el directorio corriente es `k`, entonces `".."` se refiere a `j`. Pero si el directorio actual es `i`, `".."` se refiere a `z`. Podemos también combinar varios `".."` en la misma dirección o camino (*path*). Por ejemplo, si el directorio actual es `k`, `"../.."` será una referencia a `z` y así sucesivamente. También podemos usar el directorio padre para navegar entre los directorios. Por ejemplo, para acceder a `h`, estando en `k`, podemos escribir `"../../h"`. También podemos crear directorios en los programas utilizando la función `makedirs` (Lista 9.13).

►Lista 9.13 – Creación de directorios

```
import os
os.mkdir("d")
os.mkdir("e")
os.mkdir("f")
print(os.getcwd())
```

```
os.chdir("d")
print(os.getcwd())
os.chdir("../e")
print(os.getcwd())
os.chdir("..")
print(os.getcwd())
os.chdir("f")
print(os.getcwd())
```

La función `mkdir` crea solo un directorio por vez. Si necesita crear un directorio, sabiendo o no si los superiores fueron creados, use la función `makedirs`, que crea todos los directorios intermedios de una sola vez.

►Lista 9.14 – Creación de directorios intermedios de una sola vez

```
import os
os.makedirs("abuelo/padre/hijo")
os.makedirs("abuelo/madre/hija")
```

Abra el directorio actual usando Windows Explorer, o Finder en el Mac OS X, para ver los directorios creados (Figura 9.2).

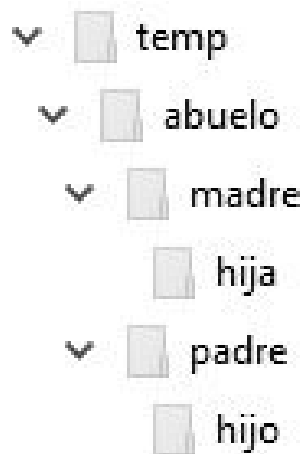


Figura 9.2 – Estructura de directorios creada en disco en Windows Explorer de Windows 8.

Puede cambiar el nombre de un directorio o archivo; esto quiere decir que puede renombrarlo usando la función `rename` (Lista 9.15).

►Lista 9.15 – Alteración del nombre de archivos y directorios

```
import os
os.mkdir("viejo")
os.rename("viejo", "nuevo")
```

La función `rename` también puede ser utilizada para mover archivos, para lo cual basta especificar el mismo nombre en otro directorio (Lista 9.16).

►Lista 9.16 – Alteración del nombre de archivos y directorios


```
import os
os.makedirs("abuelo/padre/hijo")
os.makedirs("abuelo/madre/hija")
os.rename("abuelo/padre/hijo","abuelo/madre/hijo")
```

Si quiere borrar un directorio, utilice la función `rmdir`. Si quiere borrar un archivo, use la función `remove` (Lista 9.17).

►Lista 9.17 – Exclusión de archivos y directorios

```
import os
# Crea un archivo y lo cierra inmediatamente
open("moribundo.txt","w").close()
os.mkdir("vacío")
os.rmdir("vacío")
os.remove("moribundo.txt")
```

Podemos también solicitar una lista de todos los archivos y directorios usando la función `listdir`. Los archivos y directorios serán retornados como elementos de la lista (Lista 9.18).

►Lista 9.18 – Lista del nombre de archivos y directorios

```
import os
print(os.listdir("."))
print(os.listdir("abuelo"))
print(os.listdir("abuelo/padre"))
print(os.listdir("abuelo/madre"))
```

Donde "." significa el directorio actual. La lista contiene solo el nombre de cada archivo. Vamos a ver cómo obtener el tamaño del archivo y las fechas de creación, acceso y modificación a continuación, con el módulo `os.path`.

El módulo `os.path` trae varias otras funciones que vamos a utilizar para obtener más informaciones sobre los archivos en disco. Las dos primeras son `isdir` y `isfile`, que retornan `True` si el nombre pasado es un directorio o un archivo respectivamente (Lista 9.19).

►Lista 9.19 – Verificando si es directorio o archivo

```
import os
import os.path
for a in os.listdir("."):
    if os.path.isdir(a):
        print("%s/" % a)
    elif os.path.isfile(a):
        print("%s" % a)
```

Ejecute el programa y vea que imprimimos solo los nombres de directorios y archivos, ya que

adicionamos una barra en el final de los nombres de directorio.

Podemos también verificar si un directorio o archivo ya existe con la función **exists** (Lista 9.20).

► Lista 9.20 – Verificando si un directorio o archivo ya existe

```
import os.path
if os.path.exists("z"):
    print("El directorio z existe.")
else:
    print("El directorio z no existe.")
```

Ejercicio 9.31 Cree un programa que corrija el de la lista 9.20 de modo de verificar si **z** existe y es un directorio.

Ejercicio 9.32 Modifique el programa de la lista 9.20 de modo de recibir el nombre del archivo o directorio a verificar por la línea de comando. Imprima si existe y si es un archivo o un directorio.

Ejercicio 9.33 Cree un programa que genere una página html con links para todos los archivos jpg y png encontrados a partir de un directorio informado en la línea de comando.

Tenemos también otras funciones que retornan más informaciones sobre archivos y directorios, tales como su tamaño y fechas de modificación, creación y acceso (Lista 9.21).

► Lista 9.21 – Obtención de más informaciones sobre el archivo

```
import os
import os.path
import time
import sys
nombre = sys.argv[1]
print("Nombre: %s" % nombre)
print("Tamaño: %d" % os.path.getsize(nombre))
print("Creado: %s" % time.ctime(os.path.getctime(nombre)))
print("Modificado: %s" % time.ctime(os.path.getmtime(nombre)))
print("Accedido: %s" % time.ctime(os.path.getatime(nombre)))
```

Donde **getsize** retorna el tamaño del archivo en bytes, **getctime** retorna la fecha y hora de creación, **getmtime** de modificación y **getatime** de acceso. Observe que llamamos **time.ctime** para transformar la fecha y hora retornadas por **getmtime**, **getatime** y **getctime** en cadena de caracteres. Esto es necesario porque el valor retornado es expresado en segundos y necesita ser correctamente convertido para ser exhibido.

9.7 Un poco sobre el tiempo

El módulo **time** trae varias funciones para manipular el tiempo. Una de ellas fue presentada en la

sección anterior: `time.ctime`, que convierte un valor en segundos después de 01/01/1970 en cadena de caracteres. Tenemos también la función `gmtime`, que retorna una tupla con componentes del tiempo separados en elementos. Veamos algunos ejemplos en el intérprete (Lista 9.22).

► **Lista 9.22 – Obtención de las horas en Python**

```
>>>import time
>>> ahora=time.time()
>>> ahora
1277310220.906508
>>> time.ctime(ahora)
'Wed Jun 23 18:23:40 2010'
>>> ahora2=time.localtime()
>>> ahora2
time.struct_time(tm_year=2010, tm_mon=6, tm_mday=23, tm_hour=18, tm_min=23,
tm_sec=40, tm_wday=2, tm_yday=174, tm_isdst=1)
>>> time.gmtime(ahora)
time.struct_time(tm_year=2010, tm_mon=6, tm_mday=23, tm_hour=16, tm_min=23,
tm_sec=40, tm_wday=2, tm_yday=174, tm_isdst=0)
```

La función `time.time` devuelve la hora actual en segundos, usando el horario de Greenwich o UTC (Tiempo Universal Coordinado). En la lista 9.22, atribuimos su resultado a la variable `ahora` y lo convertimos en cadena de caracteres usando `time.ctime`. Si desea trabajar con la hora en su huso horario, utilice `time.localtime`, como se muestra con la variable `ahora2`. Observe que `time.localtime` devolvió una tupla y que `time.time` retorna solo un número. Podemos utilizar la función `gmtime` para convertir la variable `ahora` en una tupla de nueve elementos, como la retornada por `time.localtime` (Tabla 9.2).

Tabla 9.2 – Elementos de la tupla de tiempo retornada por gmtime

Posición	Nombre	Descripción
0	<code>tm_year</code>	año
1	<code>tm_mon</code>	mes
2	<code>tm_mday</code>	día
3	<code>tm_hour</code>	hora
4	<code>tm_min</code>	minutos
5	<code>tm_sec</code>	segundos
6	<code>tm_wday</code>	día de la semana entre 0 y 6, donde lunes es 0
7	<code>tm_yday</code>	día del año, varía de 1 a 366.
8	<code>tm_isdst</code>	horario de verano, donde 1 indica que es el horario de verano.

Podemos también acceder a esos datos por nombre (Lista 9.23).

Lista 9.23 – Obtención de fecha y hora por nombre

```
import time
ahora=time.localtime()
print("Año: %d" % ahora.tm_year)
print("Mes: %d" % ahora.tm_mon)
print("Día: %d" % ahora.tm_mday)
print("Hora: %d" % ahora.tm_hour)
print("Minuto: %d" % ahora.tm_min)
print("Segundo: %d" % ahora.tm_sec)
print("Día de la semana: %d" % ahora.tm_wday)
print("Día del año: %d" % ahora.tm_yday)
print("Horario de verano: %d" % ahora.tm_isdst)
```

La función `time.strftime` permite formatear el tiempo en cadena de caracteres. Puede pasar el formato deseado para la cadena de caracteres, siguiendo los códigos de formateo de la tabla 9.3.

Tabla 9.3 – Códigos de formateo de strftime

Código	Descripción
%a	día de la semana abreviado
%A	nombre del día de la semana
%b	nombre del mes abreviado
%B	nombre del mes completo
%c	fecha y hora según configuración regional
%d	día del mes (01-31)
%H	hora en el formato 24 h (00-23)
%I	hora en el formato 12 h
%j	día del año 001-366
%m	mes (01-12)
%M	minutos (00-59)
%p	AM o PM
%S	segundos (00-61)
%U	número de la semana (00-53), donde la semana 1 comienza después del primer domingo.
%w	día de la semana (0-6) donde 0 es el domingo
%W	número de la semana (00-53), donde la semana 1 comienza después del primer lunes
%x	representación regional de la comparación
%X	representación regional de la hora
%y	año (00-99)
%Y	año con 4 dígitos
%Z	nombre del huso horario
%	símbolo de %

Si necesita convertir una tupla en segundos, utilice la función `timegm` del módulo `calendar`. Si necesita trabajar con fecha y hora en sus programas, consulte la documentación de Python sobre los módulos `time`, `datetime`, `calendar` y `locale`.

Ejercicio 9.34 Altere el programa de la lista 7.45, el juego del ahorcado. Esta vez, utilice las funciones de tiempo para cronometrar la duración de las partidas.

9.8 Uso de directorios

Una tarea común cuando se trabaja con archivos es manipular directorios. Como esa tarea depende del sistema operativo – y cada sistema tiene sus propias características, como “/” en el Linux y en el Mac OS X para separar el nombre de los directorios y “\” en Windows, la biblioteca estándar de Python trae algunas funciones interesantes. Aquí veremos las más importantes, la documentación de Python trae la lista completa. Veamos algunos ejemplos de esas funciones cuando son llamadas en el intérprete de Python, en la lista 9.24.

►Lista 9.24 – Uso de directorios

```
>>> import os.path
>>> ruta="i/j/k"
>>> os.path.abspath(ruta)
'C:\\Python31\\i\\j\\k'
>>> os.path.basename(ruta)
'k'
>>> os.path.dirname(ruta)
'i/j'
>>> os.path.split(ruta)
('i/j', 'k')
>>> os.path.splitext("archivo.txt")
('archivo', '.txt')
>>> os.path.splitdrive("c:/Windows")
('c:', '/Windows')
```

La función `abspath` devuelve la ruta de acceso absoluta del directorio o archivo pasado como parámetro. Si la ruta no empieza con `/`, el directorio actual es acrecentado, retornando la ruta completa a partir de la raíz. En el caso de Windows, incluye también la letra del disco (*drive*), en el caso `C:`.

Observe que la ruta “`j/k`” es solo una cadena de caracteres. Las funciones de `os.path` no verifican si esa ruta realmente existe. En realidad, `os.path`, la mayoría de las veces ofrece solo funciones inteligentes para manipulación de rutas como cadenas de caracteres.

La función `basename` retorna solo la última parte de la ruta, en el ejemplo, `k`. Ya la función `dirname` devuelve la ruta a la izquierda de la última barra. Una vez más, esas funciones no verifican si `k` es un archivo o un directorio. Considere que `basename` retorna la parte de la ruta a la derecha de la última

barra, y que `dirname` retorna la ruta a la izquierda. Puede combinar el resultado de esas dos funciones con la función `split`, que retorna una tupla donde los elementos son iguales a los resultados de `dirname` y `basename`.

En Windows, puede también usar la función `splitdrive` para separar la letra del drive de la ruta en sí. La función retorna una tupla, donde la letra del drive es el primer elemento; y el resto de la dirección o ruta, el segundo.

La función `join` junta los componentes de una ruta, separándolos con barras, si es necesario. Vea el ejemplo de la lista 9.25. En Windows, la función verifica si el nombre termina con “:” y, en ese caso, no inserta una barra, permitiendo la creación de una ruta relativa. Podemos combinar el resultado de esa función con `abspath` y obtener una ruta a partir de la raíz. Vea que la manipulación de la letra del drive se hace automáticamente.

►Lista 9.25 – Combinación de los componentes de una ruta

```
>>> import os.path
>>> os.path.join("c:", "datos", "programas")
'c:datos\\programas'
>>> os.path.abspath(os.path.join("c:", "datos", "programas"))
'C:\\Python31\\datos\\programas'
```

9.9 Visita a todos los subdirectorios recursivamente

La función `os.walk` facilita la navegación en un árbol de directorios. Imagine que desee recorrer todos los directorios a partir de un directorio inicial, retornando el nombre del directorio que está siendo visitado (**raíz**); los directorios encontrados dentro del directorio siendo visitado (**directorios**) y una lista de sus archivos (**archivos**). Observe y ejecute el programa de la lista 9.26. Debe ejecutarlo pasando el directorio inicial a visitar, en la línea de comando.

►Lista 9.26 – Árbol de directorios siendo recorrido

```
import os
import sys
for raíz, directorios, archivos in os.walk(sys.argv[1]):
    print("\nRuta:", raíz)
    for d in directorios:
        print("  %s/" % d)
    for f in archivos:
        print("  %s" % f)
    print("%d directorio(s), %d archivo(s)" % (len(directorios), len(archivos)))
```

La gran ventaja de la función `os.walk` es que visita automáticamente todos los subdirectorios dentro del directorio pasado como parámetro, haciéndolo repetidamente hasta navegar completamente el árbol de directorios.

Combinando la función `os.walk` con las funciones de manipulación de directorios y archivos que ya

conocemos, puede escribir programas para manipular árboles de directorios completos.

Ejercicio 9.35 Utilizando la función `os.walk`, cree una página HTML con el nombre y tamaño de cada archivo de un directorio pasado y de sus subdirectorios.

Ejercicio 9.36 Utilizando la función `os.walk`, cree un programa que calcule el espacio ocupado por cada directorio y subdirectorio, generando una página html con los resultados.

Clases y objetos

La programación orientada a objetos facilita la escritura y mantenimiento de nuestros programas, utilizando clases y objetos. Clases es la definición de un nuevo tipo de datos que asocia datos y operaciones en una sola estructura. Un objeto puede ser entendido como una variable cuyo tipo es una clase, o sea, un objeto es una instancia de una clase.

La programación orientada a objetos es una técnica de programación que organiza nuestros programas en clases y objetos en vez de usar solo funciones, como vimos hasta ahora. Es un asunto muy importante y extenso, que merecería varios libros y mucha práctica para ser completamente entendido. El objetivo de este capítulo es presentar lo básico de la programación orientada a objetos de modo de introducir el concepto y estimular el aprendizaje de esa técnica.

10.1 Objetos como representación del mundo real

Podemos entender un objeto en Python como la representación de un objeto del mundo real, escrita en un lenguaje de programación. Esa representación es limitada por la cantidad de detalles que podemos o que queremos representar, es una abstracción.

Veamos, por ejemplo, un aparato de televisión. Podemos decir que una televisión tiene una marca y un tamaño de pantalla. Podemos también pensar en lo que podemos hacer con ese aparato, por ejemplo cambiar de canal, encenderlo o apagarlo. Veamos cómo escribir eso en Python en la lista 10.1.

► Lista 10.1 – Modelado de una televisión

```
>>>class Televisión: ❶
    def __init__(self): ❷
        self.encendida = False ❸
        self.canal = 2 ❹
>>> tv = Televisión() ❺
>>> tv.encendida ❻
False
>>> tv.canal
2
>>> tv_sala=Televisión() ❼
>>> tv_sala.encendida=True ❽
>>> tv_sala.canal=4 ❾
```



```
>>> tv.canal
2
>>> tv_sala.canal
4
```

En ❶, creamos una nueva clase llamada **Televisión**. Utilizamos la instrucción **class** para indicar la declaración de una nueva clase y ":" para comenzar su bloque. Cuando declaramos una clase, estamos creando un nuevo tipo de datos. Ese nuevo tipo define sus propios métodos y atributos. Recuerde los tipos cadena de caracteres y **list**. Esos dos tipos predefinidos de Python son clases. Cuando creamos una lista o una cadena de caracteres, estamos instanciando o creando una instancia de esas clases, o sea, un objeto. Cuando definimos nuestras propias clases, podemos crear nuestros propios métodos y atributos.

En ❷, definimos un método especial llamado **__init__**. Los métodos son funciones asociadas a una clase. El método **__init__** será llamado siempre que cree objetos de la clase **Televisión**, siendo por eso llamado constructor. Un método constructor es llamado siempre que un objeto de la clase es instanciado. Es el constructor el que inicializa nuestro nuevo objeto con sus valores estándar. El método **__init__** recibe un parámetro llamado **self**. Por ahora, entienda **self** como el objeto televisión en sí, lo que quedará más claro más adelante en este libro.

En ❸, decimos que **self.encendida** es un valor de **self**, o sea, del objeto televisión. Todo método en Python tiene **self** como primer parámetro. Decimos que **self.encendida** es un atributo del objeto. Como **self** representa el objeto en sí, escribiremos **self.encendida**. Siempre que queremos especificar atributos de objetos, debemos asociarlos a **self**. En caso contrario, si escribiésemos **encendida=False**, encendida sería solo una variable local del método **__init__**, y no un atributo del objeto.

En ❹, decimos que **canal** también es un valor o característica de nuestra televisión. Observe también que escribimos **self.canal** para crear un atributo, y no una simple variable local.

En ❺ creamos un objeto **tv** utilizando la clase **Televisión**. Decimos que **tv** es ahora un objeto de la clase **Televisión** o que **tv** es una instancia de **Televisión**. Cuando solicitamos la creación de un objeto, el método constructor de su clase es llamado, en Python, **__init__**, como declaramos en ❷.

En ❻, exhibimos el valor del atributo **encendida** y **canal** del objeto **tv**.

Ya en ❼ creamos otra instancia de la clase **Televisión** llamada **tv_sala**. En ❽, cambiamos el valor de encendida para **True** y el canal para 4 en ❾.

Observe que al imprimir el canal de cada televisión tenemos valores independientes, pues **tv** y **tv_sala** son dos objetos independientes, pudiendo cada uno tener sus propios valores, como dos televisiones en el mundo real. Cuando creamos un objeto de una clase, éste tiene todos los atributos y métodos que especificamos al declarar la clase y que fueron inicializados en su constructor. Esa característica simplifica el desarrollo de los programas, pues podemos definir el comportamiento de todos los objetos de una clase (métodos), preservando los valores individuales de cada uno (atributos).

Ejercicio 10.1 Adicione los atributos tamaño y marca a la clase **Televisión**. Cree dos objetos **Televisión** y atribuya tamaños y marcas diferentes. Después, imprima el valor de esos atributos de modo de confirmar la

Veamos ahora cómo asociar un comportamiento a la clase Televisión, definiendo dos métodos `cambia_canal_para_arriba` y `cambia_canal_para_abajo` (Lista 10.2).

► Lista 10.2 – Adición de métodos para cambiar el canal

```
>>>class Televisión:
...     def __init__(self):
...         self.encendida=False
...         self.canal=2
...     def cambia_canal_para_abajo(self): ❶
...         self.canal-=1
...     def cambia_canal_para_arriba(self): ❷
...         self.canal+=1
...
>>> tv = Televisión()
>>> tv.cambia_canal_para_arriba() ❸
>>> tv.cambia_canal_para_arriba()
>>> tv.canal
4
>>> tv.cambia_canal_para_abajo() ❹
>>> tv.canal
3
```

En ❶, definimos el método `cambia_canal_para_abajo`. Observe que no utilizamos `"_"` antes del nombre del método, pues ese nombre no es un nombre especial de Python, sino solo un nombre elegido por nosotros. Vea que pasamos también un parámetro `self`, que representa el objeto en sí. Observe que escribimos directamente `self.canal-=1`, utilizando el atributo `canal` de la televisión. Esto es posible porque creamos el atributo `canal` en el constructor(`__init__`). Es usando atributos que podemos almacenar valores entre las llamadas de los métodos.

En ❷, hicimos lo mismo, pero, esta vez, con `cambia_canal_para_arriba`.

En ❸, llamamos al método. Observe que escribimos el nombre del método después del nombre del objeto, separándolos con un punto, de tal manera que el método fue llamado del mismo modo que una función, pero en la llamada no pasamos ningún parámetro. En realidad, el intérprete de Python adiciona el objeto `tv` a la llamada, utilizándolo como el `self` del método en ❷. Es así que el intérprete logra trabajar con varios objetos de una misma clase.

Después, en ❹, hacemos la llamada del método `cambia_canal_para_abajo`. Vea el valor retornado por `tv.canal`, antes y después de llamar al método.

La gran ventaja de usar clases y objetos es facilitar la construcción de los programas. Aunque simple, puede observar que no necesitamos enviar el canal actual de la televisión al método `cambia_canal_para_arriba`, simplificando la llamada del método. Ese efecto “memoria” facilita la

configuración de objetos complejos, pues almacenamos las características importantes en sus atributos, evitando pasar esos valores a cada llamada.

En verdad, ese tipo de construcción imita el comportamiento del objeto en el mundo real. ¡Cuando cambiamos el canal de la **tv** para arriba o para abajo, no le informamos el canal actual a la televisión!

10.2 Pasaje de parámetros

Un problema con la clase televisión es que no controlamos los límites de nuestros canales. En realidad, podemos hasta obtener canales negativos o números muy grandes, como 35790. Vamos a modificar el constructor de modo de recibir el canal mínimo y máximo soportado por **tv** (Lista 10.3).

Ejecute el programa de la lista 10.3 y verifique si las modificaciones dieron resultado. Observe que cambiamos el comportamiento de la clase **Televisión** sin cambiar casi nada en el programa que la utiliza. Eso es porque aislamos los detalles del funcionamiento de la clase del resto del programa. Ese efecto es llamado de encapsulamiento. Decimos que una clase debe encapsular u ocultar detalles de su funcionamiento el máximo posible. En el caso de los métodos para cambiar el canal, incluimos la verificación sin alterar el resto del programa. Simplemente solicitamos que el método realice su trabajo, sin preocuparse por los detalles internos de cómo realizará esa operación.

► Lista 10.3 – Verificación de la franja de canales de tv

```
class Televisión:
    def __init__(self, min, max):
        self.encendida = False
        self.canal = 2
        self.cmin = min
        self.cmax = max
    def cambia_canal_para_abajo(self):
        if(self.canal-1>=self.cmin):
            self.canal-=1
    def cambia_canal_para_arriba(self):
        if(self.canal+1<=self.cmax):
            self.canal+=1
tv=Televisión(1,99)
for x in range(0,120):
    tv.cambia_canal_para_arriba()
print(tv.canal)
for x in range(0,120):
    tv.cambia_canal_para_abajo()
print(tv.canal)
```

Ejercicio 10.2 Actualmente, la clase **Televisión** inicializa el canal con 2. Modifique la clase **Televisión** de modo

de recibir el canal inicial en su constructor.

Ejercicio 10.3 Modifique la clase **Televisión** de modo que, si pedimos para cambiar el canal para abajo, además del mínimo, ella va al canal máximo. Si cambiamos para arriba, además del canal máximo, que vuelva al canal mínimo. Ejemplo:

```
>>> tv=Televisión(2,10)
>>> tv.cambia_canal_para_abajo()
>>> tv.canal
10
>>> tv.cambia_canal_para_arriba()
>>> tv.canal
2
```

Al trabajar con clases y objetos, así como hicimos al estudiar funciones, necesitamos representar en Python una abstracción del problema. Cuando realizamos una abstracción reducimos los detalles del problema a lo necesario para solucionarlo. Estamos construyendo un modelo, o sea, modelando nuestras clases y objetos.

Antes de continuar, debe entender que el modelo puede variar de una persona a otra, como todas las partes del programa. Uno de los detalles más difíciles es decidir cuánto representar y dónde limitar los modelos. En el ejemplo de la clase **Televisión** no escribimos nada sobre el enchufe de la TV, si ésta tiene control remoto, en qué parte de la casa está ubicada o incluso si tiene control de volumen.

Como regla simple, modele solo las informaciones que necesita, adicionando detalles a medida que es necesario. Con el tiempo, la experiencia le enseñará cuándo dejar de detallar su modelo, como también apuntará errores comunes.

Todo lo que aprendimos con funciones es también válido para métodos. La principal diferencia es que un método está asociado a una clase y actúa sobre un objeto. El primer parámetro del método es llamado **self** y representa la instancia sobre la cual el método actuará. Es por medio de **self** que tendremos acceso a los otros métodos de una clase, preservando todos los atributos de nuestros objetos. No necesita pasar el objeto como primer parámetro al invocar o llamar a un método: el intérprete de Python hace eso automáticamente por usted. Entre tanto, no se olvide de declarar **self** como el primer parámetro de sus métodos.

Ejercicio 10.4 Utilizando lo que aprendimos con funciones, modifique el constructor de la clase **Televisión** de modo que **min** y **max** sean parámetros opcionales, donde **min** vale 2 y **max** vale 14, en caso que otro valor no sea pasado.

Ejercicio 10.5 Utilizando la clase **Televisión** modificada en el ejercicio anterior, cree dos instancias (objetos), especificando el valor de **min** y **max** por nombre.

10.3 Ejemplo de un banco

Veamos un ejemplo de clases, pero esta vez vamos a modelar cuentas corrientes de un banco.

Imagine el Banco Tatu, moderno y eficiente, pero necesitado de un nuevo programa para controlar el saldo de sus cuentacorrentistas. Cada cuenta corriente puede tener uno o más clientes como titular. El banco controla solo el nombre y teléfono de cada cliente. La cuenta corriente presenta un saldo y una lista de operaciones de extracciones y depósitos. Cuando el cliente haga una extracción, disminuirémos el saldo de la cuenta corriente. Cuando haga un depósito, aumentaremos el saldo. Por ahora, el Banco Tatu no ofrece cuentas especiales, o sea que el cliente no puede sacar más dinero de lo que su saldo le permite.

Vamos a resolver este problema por partes. La clase **Cliente** es simple, tiene solo dos atributos: nombre y teléfono. Digite el programa de la lista 10.4 y guárdelo en un archivo llamado *clientes.py*.

► Lista 10.4 – Clase Clientes (clientes.py)

class Cliente:

```
def __init__(self, nombre, teléfono):
    self.nombre = nombre
    self.teléfono = teléfono
```

Abra el archivo *clientes.py* en el IDLE y ejecútelo (Run – F5). En el intérprete, pruebe crear un objeto de la clase **Cliente**.

```
juan=Cliente("Juan da Silva", "777-1234")
maría=Cliente("María Silva", "555-4321")
juan.nombre
juan.teléfono
maría.nombre
maría.teléfono
```

Como en los otros programas de Python, podemos ejecutar una definición de clase dentro de un archivo *.py* y utilizar el intérprete para probar nuestras clases y objetos. Esa operación es muy importante para probar las clases, modificar algunos valores y repetir el test.

Como el programa del Banco Tatu va a ser mayor que los programas que ya trabajamos hasta aquí, vamos a grabar cada clase en un archivo *.py* separado. En Python esa organización no es obligatoria, pues el lenguaje permite que tengamos todas nuestras clases en un solo archivo, si así lo queremos.

Vamos ahora a crear el archivo *test.py*, que simplemente va a importar *clientes.py* y crear dos objetos.

► Lista 10.5 – Programa test.py que importa la clase Cliente (clientes.py)

from clientes **import** Cliente

```
juan=Cliente("Juan da Silva", "777-1234")
maría=Cliente("María da Silva", "555-4321")
```

Observe que con pocas líneas de código conseguimos reutilizar la clase **Cliente**. Esto es posible porque *clientes.py* y *test.py* están en el mismo directorio. Esa separación permite que pasemos a probar las nuevas clases en el intérprete, almacenando el **import** y la creación de los objetos en un archivo aparte. Así podremos definir nuestras clases separadamente de los experimentos o pruebas.

Para resolver el problema del Banco Tatu, necesitamos otra clase, **Cuenta**, para representar una cuenta del banco con sus clientes y su saldo. Veamos el programa de la lista 10.6. La clase **Cuenta** es definida recibiendo clientes, número y saldo en su constructor (`__init__`), donde en clientes esperamos una lista de objetos de la clase **Cliente**; número es una cadena de caracteres con el número de la cuenta; y saldo es un parámetro opcional, teniendo cero (0) como estándar. La lista también presenta los métodos **resumen**, **extracción** y **depósito**. El método **resumen** exhibe en la pantalla el número de la cuenta corriente y su saldo. **extracción** permite retirar dinero de la cuenta corriente, verificando si esa operación es posible (`self.saldo >= valor`). **depósito** simplemente adiciona el valor solicitado al saldo de la cuenta corriente.

► Lista 10.6 – Clase Cuenta (cuentas.py)

```
class Cuenta:
    def __init__(self, clientes, número, saldo = 0):
        self.saldo = saldo
        self.clientes = clientes
        self.número = número
    def resumen(self):
        print("CC Número: %s Saldo: %10.2f" %
              (self.número, self.saldo))
    def extracción(self, valor):
        if self.saldo >= valor:
            self.saldo -= valor
    def depósito(self, valor):
        self.saldo += valor
```

Altere el programa *test.py* de modo de importar la clase **Cuenta**. Cree una cuenta corriente para los clientes Juan y María.

Haga algunas pruebas en el intérprete:

```
cuenta.resumen()
cuenta.extracción(1000)
cuenta.resumen()
cuenta.extracción(50)
cuenta.resumen()
cuenta.depósito(200)
cuenta.resumen()
```

Aunque nuestra cuenta corriente empiece a funcionar, aún no tenemos la lista de operaciones de cada elemento. Esa lista es, en realidad, un extracto de cuenta. Vamos a alterar la clase **Cuenta** de modo de adicionar un atributo que es la lista de operaciones realizadas (Lista 10.7). Considere el saldo inicial como un depósito. Vamos a adicionar también un método **extracción** para imprimir todas las operaciones realizadas.

► Lista 10.7 – Cuenta con registro de operaciones y extracción (cuentas.py)

```
class Cuenta:
    def __init__(self, clientes, número, saldo = 0):
        self.saldo = 0
        self.clientes = clientes
        self.número = número
        self.operaciones = []
        self.depósito(saldo)
    def resumen(self):
        print("CC N°%s Saldo: %10.2f" %
            (self.número, self.saldo))
    def extracción(self, valor):
        if self.saldo >= valor:
            self.saldo -=valor
            self.operaciones.append(["EXTRACCIÓN", valor])
    def depósito(self, valor):
        self.saldo += valor
        self.operaciones.append(["DEPÓSITO", valor])
    def extracto(self):
        print("Extracto CC N° %s\n" % self.número)
        for o in self.operaciones:
            print("%10s %10.2f" % (o[0],o[1]))
        print("\n    Saldo: %10.2f\n" % self.saldo)
```

Modifique también el programa de pruebas para imprimir la extracción de cada cuenta (Lista 10.8).

► Lista 10.8 – Probando Cliente y Cuentas

```
from clientes import Cliente
from cuentas import Cuenta
juan=Cliente("Juan da Silva", "777-1234")
maría=Cliente("María da Silva", "555-4321")
cuenta1=Cuenta([juan], 1, 1000)
cuenta2=Cuenta([maría, juan], 2, 500)
cuenta1.extracción(50)
cuenta2.depósito(300)
cuenta1.extracción(190)
cuenta2.depósito(95.15)
cuenta2.extracción(250)
cuenta1.extracto()
```

cuenta2.extracto()

Ejercicio 10.6 Altere el programa de modo que el mensaje saldo insuficiente sea exhibido en caso que haya intento de sacar más dinero que el saldo disponible.

Ejercicio 10.7 Modifique el método resumen de la clase **Cuenta** para exhibir el nombre y el teléfono de cada cliente.

Ejercicio 10.8 Cree una nueva cuenta, ahora teniendo a Juan y José como clientes y saldo igual a 500.

Para resolver el problema del Banco Tatu necesitamos una clase para almacenar todas nuestras cuentas. Como atributos del banco tendríamos su nombre y la lista de cuentas. Como operaciones, considere la apertura de una cuenta corriente y la lista de todas las cuentas del banco.

► Lista 10.9 – Clase Banco (bancos.py)

```
class Banco:
    def __init__(self, nombre):
        self.nombre=nombre
        self.clientes=[]
        self.cuentas=[]
    def abre_cuenta(self, cuenta):
        self.cuentas.append(cuenta)
    def lista_cuentas(self):
        for c in self.cuentas:
            c.resumen()
```

Ahora, vamos crear los objetos (Lista 10.10).

► Lista 10.10 – Creando los objetos

```
from clientes import Cliente
from bancos import Banco
from cuentas import Cuenta
juan = Cliente("Juan da Silva", "3241-5599")
maría = Cliente("María Silva", "7231-9955")
josé = Cliente("José Vargas","9721-3040")
cuentaJM = Cuenta( [juan, maría], 100)
cuentaJ = Cuenta( [josé], 10)
tatu = Banco("Tatú")
tatu.abre_cuenta(cuentaJM)
tatu.abre_cuenta(cuentaJ)
tatu.lista_cuentas()
```


Ejercicio 10.9 Cree clases para representar estados y ciudades. Cada estado tiene un nombre, sigla y ciudades. Cada ciudad tiene nombre y población. Escriba un programa de pruebas que cree tres estados con algunas ciudades en cada uno. Exhiba la población de cada estado como la suma de la población de sus ciudades.

10.4 Herencia

La orientación a objetos permite modificar nuestras clases, adicionando o modificando atributos y métodos, teniendo como base otra clase. Veamos el ejemplo del Banco Tatu, donde el nuevo sistema fue un éxito. Para atraer nuevos clientes, el Banco Tatu empezó a ofrecer cuentas especiales a los clientes. Una `CuentaEspecial` permite que podamos sacar más dinero del que está actualmente disponible en el saldo de la cuenta, hasta un determinado límite. Las operaciones de depósito, extracción y resumen son las mismas de una cuenta normal. Vamos a crear la clase `CuentaEspecial` heredando el comportamiento de la clase `Cuenta`. Digite la lista 10.11 en el mismo archivo donde la clase `Cuenta` fue definida (`cuentas.py`).

► Lista 10.11 – Uso de herencia para definir `CuentaEspecial`

```
class CuentaEspecial(Cuenta): ❶
    def __init__(self, clientes, número, saldo = 0, límite=0):
        Cuenta.__init__(self, clientes, número, saldo) ❷
        self.límite = límite ❸
    def extracción(self, valor):
        if self.saldo + self.límite >= valor:
            self.saldo -= valor
            self.operaciones.append(["EXTRACCIÓN", valor])
```

En ❶ definimos la clase `CuentaEspecial`, pero observe que escribimos `Cuenta` entre paréntesis. Ese es el formato de declaración de clase usando herencia; es así que declaramos la herencia de una clase en Python. Esa línea dice: cree una nueva clase llamada `CuentaEspecial` heredando todos los métodos y atributos de la clase `Cuenta`. A partir de aquí, `CuentaEspecial` es una subclase de `Cuenta`. También decimos que `Cuenta` es la superclase de `CuentaEspecial`.

En ❷, llamamos el método `__init__` de `Cuenta`, escribiendo `Cuenta.__init__` seguido de los parámetros que normalmente pasaríamos. Cada vez que usted utiliza herencia, el método constructor de la superclase, en este caso `Cuenta`, debe ser llamado. Observe que pasamos `self` para `Cuenta.__init__`. Es así que reutilizamos las definiciones ya realizadas en la superclase, evitando tener que reescribir los atributos de clientes, número y saldo. Llamar la inicialización de la superclase también tiene otras ventajas, como garantizar que las modificaciones en el constructor de la superclase no tengan que ser duplicadas en todas las subclases.

En ❸ creamos el atributo `self.límite`. Ese atributo será creado solo para clases del tipo `CuentaEspecial`. Observe que creamos el nuevo atributo después de llamar `Cuenta.__init__`.

Observe también que no llamamos `Cuenta.extracción` en el método `extracción` de `CuentaEspecial`. Cuando eso ocurre, estamos sustituyendo completamente la implementación del método por una nueva. Una de las grandes ventajas de utilizar herencia de clases es justamente

poder sustituir o complementar métodos ya definidos.

Para probar esa modificación, escriba el tramo de programa de la lista 10.11 en el mismo archivo en que usted definió la clase **Cuenta**. Modifique su programa de test para que se parezca al programa de la lista 10.12.

Lista 10.12 – Creación y uso de una **CuentaEspecial**

```
from clientes import Cliente
from cuentas import Cuenta, CuentaEspecial ❶
juan=Cliente("Juan da Silva", "777-1234")
maría=Cliente("María da Silva", "555-4321")
cuenta1=Cuenta([juan], 1, 1000)
cuenta2=CuentaEspecial([maría, juan], 2, 500, 1000) ❷
cuenta1.extracción(50)
cuenta2.depósito(300)
cuenta1.extracción(190)
cuenta2.depósito(95.15)
cuenta2.extracción(1500)
cuenta1.extracto()
cuenta2.extracto()
```

Veamos qué cambió en el programa de pruebas. En ❶ adicionamos el nombre de la clase **CuentaEspecial** a **import**. De esa forma podremos utilizar la nueva clase en nuestras pruebas. Ya en ❷ creamos un objeto **CuentaEspecial**. Vea que, prácticamente, no cambiamos nada, excepto el nombre de la clase, que ahora es **CuentaEspecial** y no **Cuenta**. Un detalle importante para el test es que adicionamos un parámetro al constructor, en este caso 1000, como el valor de **límite**. Ejecute este programa de test y observe que para la **cuenta2**, obtuvimos un saldo negativo.

Utilizando herencia, modificamos muy poco nuestro programa, manteniendo la funcionalidad anterior y agregando nuevos recursos. Lo interesante de todo es que fue posible reutilizar los métodos que ya habíamos definido en la clase **Cuenta**. Eso permitió que la definición de la clase **CuentaEspecial** fuese mucho menor, pues allí especificamos solo el comportamiento que es diferente.

Cuando utilice herencia, trate de crear clases en las cuales el comportamiento y características comunes estén en la superclase. De esa forma podrá definir subclases solo con lo necesario. Otra ventaja de utilizar herencia es que si cambiamos algo en la superclase, esos cambios serán también usados por las subclases. Un ejemplo sería modificar el método de **extracto**. Como en **CuentaEspecial** no especificamos un nuevo método de **extracto**, al modificar el método **Cuenta.extracto**, estaremos también modificando el **extracto** de **CuentaEspecial**, pues las dos clases comparten el mismo método.

Es importante notar que, al utilizar herencia, las subclases deben poder sustituir sus superclases, sin pérdida de funcionalidad y sin generar errores en los programas. Lo importante es que conozca ese nuevo recurso y empiece a utilizarlo en sus programas. Recuerde que usted no está obligado a

definir una jerarquía de clases en todos sus programas. Con el tiempo, la necesidad de utilizar herencia quedará más clara.

Ejercicio 10.10 Modifique las clases `Cuenta` y `CuentaEspecial` para que la operación de extracción retorne verdadero si la extracción fue efectuada y falso en caso contrario.

Ejercicio 10.11 Altere la clase `CuentaEspecial` de modo que su extracto exhiba el límite y el total disponible para extracción.

Ejercicio 10.12 Observe el método extracción de las clases `Cuenta` y `CuentaEspecial`. Modifique el método extracción de la clase `Cuenta` de modo que la verificación de la posibilidad de extracción sea hecha por un nuevo método, sustituyendo la condición actual. Ese nuevo método retornará verdadero si la extracción puede ser efectuada, y falso en caso contrario. Modifique la clase `CuentaEspecial` de modo de trabajar con ese nuevo método. Verifique si aún necesita cambiar el método extracción de `CuentaEspecial` o solo el nuevo método creado para verificar la posibilidad de extracción.

10.5 Desarrollando una clase para controlar listas

Ahora que ya tenemos una idea de cómo utilizar clases en Python, vamos crear una clase que controle una lista de objetos. En los ejemplos anteriores, al usar la lista de clientes como una lista simple, no hicimos ninguna verificación en cuanto a la duplicación de valores. Por ejemplo, una lista de clientes donde los dos clientes son la misma persona sería aceptada sin problemas. Otro problema de nuestra lista simple es que no verifica si los elementos son objetos de la clase `Cliente`. Vamos modificar eso construyendo una nueva clase, llamada `ListaÚnica`.

► Lista 10.13 – Clase `ListaÚnica` (`listaunica.py`)

```
class ListaÚnica:
def __init__(self, elem_class):
    self.lista = []
    self.elem_class = elem_class
def __len__(self):
    return len(self.lista)
def __iter__(self):
    return iter(self.lista)
def __getitem__(self, p):
    return self.lista[p]
def indiceVálido(self, i):
    return i >= 0 and i < len(self.lista)
def adiciona(self, elem):
    if self.investigación(elem) == -1:
        self.lista.append(elem)
def remove(self, elem):
```

```

    self.lista.remove(elem)
def investigación(self, elem):
    self.verifica_tipo(elem)
    try:
        return self.lista.index(elem)
    except ValueError:
        return -1
def verifica_tipo(self, elem):
    if type(elem)!=self.elem_class:
        raise TypeError("Tipo inválido")
def ordena(self, clave=None):
    self.lista.sort(key=clave)

```

Veamos qué podemos hacer con esa clase, realizando algunas pruebas en el intérprete:

```

>>> from listaunica import *
>>> lu = ListaÚnica(int) ❶
>>> lu.adiciona(5) ❷
>>> lu.adiciona(3)
>>> lu.adiciona(2.5) ❸

```

Traceback (most recent call last):

```

  File "<stdin>", line 1, in <module>
    File "listaunica.py", line 19, in adiciona
if self.investigación(elem) == -1:
    File "listaunica.py", line 26, in investigación
self.verifica_tipo(elem)
    File " listaunica.py", line 34, in verifica_tipo
raise TypeError("Tipo inválido")
TypeError: Tipo inválido

```

En ❶ creamos un objeto con la clase **ListaÚnica** que acabamos de importar. Vea que pasamos **int** como parámetro en el constructor, indicando que esa lista debe contener solo valores del tipo **int** (enteros). En ❷, adicionamos el número entero 5 como elemento de nuestra lista, así como 3 en la línea siguiente. Vea que no tuvimos ningún problema, pero que en ❸, al tratar de adicionar 2.5 (un número del tipo **float**), obtuvimos una excepción con mensaje de error tipo inválido (**TypeError**). Eso sucede porque, en nuestro constructor, el parámetro **elem** es en realidad la clase que deseamos para nuestros elementos. Siempre que un elemento es adicionado (método **adiciona**), una verificación del tipo del nuevo elemento es hecha por el método **verifica_tipo**, que genera una excepción, en caso que el tipo no sea igual al tipo pasado al constructor. Vea que el método **adiciona** también realiza una investigación para verificar si un elemento igual ya no forma parte de la lista y solo realiza la adición en caso que un elemento igual no haya sido encontrado. De esa forma, podemos garantizar que nuestra lista contendrá solo elementos del mismo tipo y sin

repeticiones.

Vamos a continuar probando nuestra nueva clase en el intérprete:

```
>>> len(lu) ❹
2
>>> for e in lu: ❺
...     print(e)
...
5
3
>>> lu.adiciona(5) ❻
>>> len(lu) ❼
2
>>> lu[0] ❽
5
>>> lu[1]
3
```

En ❹, utilizamos la función `len` con nuestro objeto y obtuvimos el resultado esperado. Vea que no tuvimos que escribir `len(lu.lista)`, sino solo `len(lu)`. Esto es posible pues implementamos el método `__len__`, que es responsable de retornar el número de elementos a partir de `self.lista`.

En ❺, utilizamos nuestro objeto `lu` en un `for`. Eso fue posible con la implementación del método `__iter__`, que es llamado cuando utilizamos un objeto con `for`. El método `__iter__` simplemente llama la función `iter` de Python con nuestra lista interna `self.lista`. La intención de utilizar esos métodos es esconder algunos detalles de nuestra clase `ListaÚnica` y evitar el acceso directo a la `self.lista`.

En ❻, probamos la inclusión de un elemento repetido para—en la línea siguiente—, en ❼, confirmar que la cantidad de elementos de la lista no fue alterada. Eso sucede porque al realizar la investigación en el método `adiciona` el valor es encontrado, y la adición es ignorada.

En ❽, utilizamos índices con nuestra clase, del mismo modo que hacemos con una lista normal de Python. Esto es posible pues implementamos el método `__getitem__`, que recibe el valor del índice (`p`) y retorna el elemento correspondiente de nuestra lista.

Python tiene varios métodos mágicos, métodos especiales que tienen el formato `__nombre__`. El método `__init__`, usado en nuestros constructores es un método mágico, `__len__`, `__getitem__` e `__iter__` también. Esos métodos permiten dar otro comportamiento a nuestras clases y usarlas casi como clases propias del lenguaje. La utilización de esos métodos mágicos no es obligatoria, pero posibilita una gran flexibilidad para nuestras clases.

Veamos otro ejemplo de clase con métodos mágicos (especiales). La clase `Nombre`, definida en la lista 10.14, es utilizada para guardar nombres de personas y mantener una clave de investigación.

► Lista 10.14 – Clase `Nombre` (`nombre.py`)

```
class Nombre:
```

```

def __init__(self, nombre):
    if nombre == None or not nombre.strip():
        raise ValueError("Nombre no puede ser nulo ni en blanco")
    self.nombre = nombre
    self.clave = nombre.strip().lower()
def __str__(self):
    return self.nombre
def __repr__(self):
    return "<Clase {3} en 0x{0:x} Nombre: {1} Clave: {2}>".format(id(self),
                                                                self.nombre, self.clave, type(self).__name__)
def __eq__(self, otro):
    print("__eq__ Llamado")
    return self.nombre == otro.nombre
def __lt__(self, otro):
    print("__lt__ Llamado")
    return self.nombre < otro.nombre

```

En la clase **Nombre**, definimos el método `__init__` de modo de generar una excepción en caso que la cadena de caracteres pasada como nombre sea nula (**None**) o en blanco. Vea que implementamos también el método mágico `__str__`, que es llamado al imprimir un objeto de la clase **Nombre**. De esa forma podemos configurar la salida de nuestras clases con más libertad, y sin cambiar el comportamiento esperado de una clase normal en Python.

Veamos el resultado de ese programa en el intérprete:

```

>>>from nombre import *
>>> A=Nombre("Nilo") ❶
>>> print(A) ❷
Nilo
>>> B=Nombre(" ") ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "nombre.py", line 4, in __init__
raise ValueError("Nombre no puede ser nulo ni en blanco")
ValueError: Nombre no puede ser nulo ni en blanco
>>> C=Nombre(None) ❹
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "nombre.py", line 4, in __init__
raise ValueError("Nombre no puede ser nulo ni en blanco")
ValueError: Nombre no puede ser nulo ni en blanco

```

En ❶, creamos un objeto de la clase **Nombre**, pasando la cadena de caracteres "Nilo". En ❷, imprimimos el objeto A directamente, vea que el resultado fue el nombre Nilo, que es el valor retornado por nuestro método `__str__`. En ❸ y ❹, probamos la inicialización de un objeto con valores inválidos. Vea que una excepción del tipo **ValueError** fue generada, impidiendo la creación del objeto con el valor inválido. El objetivo de ese tipo de validación es garantizar que el valor usado en la clase sea válido. Podemos implementar reglas más complejas dependiendo del objetivo de nuestro programa, pero para ese ejemplo, verificar si la cadena de caracteres está en blanco o si es nula (**None**) es suficiente.

Vamos a continuar descubriendo los otros métodos especiales en el intérprete:

```
>>> A=Nombre("Nilo")
>>> A ❶
<Clase Nombre en 0x26ee8f0 Nombre: Nilo Clave: nilo>
>>> A == Nombre("Nilo") ❷
__eq__ Llamado
True
>>> A != Nombre("Nilo") ❸
__eq__ Llamado
False
>>> A < Nombre("Nilo") ❹
__lt__ Llamado
False
>>> A > Nombre("Nilo") ❺
__lt__ Llamado
False
```

En ❶, tenemos la representación del objeto A, o sea, la forma que es usada por el intérprete para mostrar el objeto fuera de la función **print** o cuando usamos la función **repr**. Vea que la salida fue generada por el método `__repr__` de la clase **Nombre**.

En ❷, utilizamos el operador `==` para verificar si el objeto A es igual a otro objeto. En Python, el comportamiento estándar es que `==` retorne **True** si los dos objetos son el mismo objeto, o **False** en caso contrario. Mientras tanto, el objeto A y **Nombre("Nilo")** son claramente dos instancias diferentes de la clase **Nombre**. Vea que el resultado es **True** y que nuestro método `__eq__` fue llamado. Eso sucede porque `__eq__` es el método especial utilizado para comparaciones de igualdad (`==`) de nuestros objetos. En nuestra implementación, retornamos **True** si el contenido de **self.nombre** es igual en los dos objetos, independientemente que sea el mismo objeto o no. Vea también que en ❸, el método `__eq__` también fue llamado, aunque no hayamos definido el método `__neq__` (`!=`). Ese comportamiento viene de la implementación estándar de esos métodos, donde `__eq__` fue negado para implementar el `__neq__` inexistente. Veremos después una forma de declarar todos los operadores de comparación implementando solamente el `__eq__` y `__lt__`.

En ❹, usamos el operador `<` (menor que) para comparar los dos objetos. Vea que el método especial `__lt__` fue llamado en ese caso. Nuestra implementación del método `__lt__` utiliza la comparación

de `self.nombre` para decidir el orden de los objetos. En ❸, el operador `>` (mayor que) retorna el valor correcto, y también llama al método `__lt__`, una vez que el método `__gt__` no fue implementado, de forma análoga a lo que sucedió entre `__eq__` y `__neq__`.

Ahora vea lo que sucede cuando tratamos de utilizar los operadores `>=` (mayor o igual) y `<=` (menor o igual):

```
>>> A >= Nombre("Nilo")
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
TypeError: unorderable types: Nombre() >= Nombre()
```

```
>>> A <= Nombre("Nilo")
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
TypeError: unorderable types: Nombre() <= Nombre()
```

Eso sucede porque los operadores `>=` y `<=` llaman a los métodos `__ge__` y `__le__`, respectivamente, y ambos no fueron implementados.

Como algunas de esas relaciones no funcionan de la manera esperada, existe una forma más simple de implementar esos métodos especiales, solo con la implementación de `__eq__` y de `__lt__`. En la lista 10.15, vamos usar un recurso llamado *decorators* (adornos o decoradores).

► Lista 10.15 – Usando decorators (nombre.py)

```
from functools import total_ordering
@total_ordering
class Nombre:
    def __init__(self, nombre):
        if nombre == None or not nombre.strip():
            raise ValueError("Nombre no puede ser nulo ni en blanco")
        self.nombre = nombre
        self.clave = Nombre.CreaClave(nombre)
    def __str__(self):
        return self.nombre
    def __repr__(self):
        return "<Clase {3} en 0x{0:x} Nombre: {1} Clave: {2}>".format(id(self),
            self.nombre, self.clave, type(self).__name__)
    def __eq__(self, otro):
        print("__eq__ Llamado")
        return self.nombre == otro.nombre
    def __lt__(self, otro):
        print("__lt__ Llamado")
        return self.nombre < otro.nombre
```



```
@staticmethod
def CreaClave(nombre):
    return nombre.strip().lower()
```

El primer decorador `@total_ordering` es definido en el módulo `functools`, por eso tuvimos que importarlo en el comienzo de nuestro programa. Él es responsable de implementar, o sea, de generar el código responsable por la implementación de todos los métodos de comparación especiales a partir de `__eq__` y de `__lt__`. De esa forma, `__neq__` será la negación de `__eq__`; `__gt__`, la negación de `__lt__`; `__le__`, la combinación de `__lt__` con `__eq__`; y `__ge__`, la combinación de `__gt__` con `__eq__`, implementado, así, todos los operadores de comparación (`==`, `!=`, `>`, `<`, `>=`, `<=`). Veamos el resultado en el intérprete:

```
>>> from nombre import *
>>> A=("Nilo")
>>> A=Nombre("Nilo")
>>> A == Nombre("Nilo")
__eq__ Llamado
True
>>> A != Nombre("Nilo")
__eq__ Llamado
False
>>> A > Nombre("Nilo")
__lt__ Llamado
__eq__ Llamado
False
>>> A < Nombre("Nilo")
__lt__ Llamado
False
>>> A <= Nombre("Nilo")
__lt__ Llamado
__eq__ Llamado
True
>>> A >= Nombre("Nilo")
__lt__ Llamado
True
```

Observe que en el programa de la lista 10.15, utilizamos otro decorador `@staticmethod` antes de la definición del método `CreaClave`. Vea también que el método `CreaClave` no tiene el parámetro `self`. Ese decorador crea un método estático, o sea, un método que puede ser llamado solo con el nombre de la clase, no necesitando un objeto para ser llamado. Veamos esto en el intérprete:

```
>>> A.CreaClave("X")
'x'
```

```
>>> Nombre.CreaClave("X")  
'x'
```

Tanto la llamada de **CreaClave** con el objeto **A** como la llamada con el nombre de la clase en **Nombre.CreaClave** funcionaron como era esperado. Los métodos estáticos son utilizados para implementar métodos que no acceden o que no necesitan acceder a las propiedades de una instancia de la clase. En el caso del método **CreaClave**, solo el parámetro **nombre** es necesario para su funcionamiento. El decorador **@staticmethod** es necesario para informar al intérprete que no pase el parámetro **self** automáticamente, como hace en los métodos normales o no estáticos. La ventaja de definir métodos estáticos es agrupar ciertas funciones en nuestras clases. De esa forma, queda claro que **CreaClave** funciona en el contexto de nombres. Si fuese definido como una función fuera de una clase, su implementación podría parecer más genérica de lo que realmente es. De esta forma, los métodos estáticos ayudan a mantener el programa cohesionado y a mantener el contexto de la implementación del método.

El programa de la lista 10.15 introduce un problema de consistencia de datos. Vea que al construir el objeto en **__init__**, verificamos el valor de **nombre** y también actualizamos la clave de investigación. El problema es que podemos alterar **nombre** sin ninguna validación y dejar clave en un estado inválido. Veamos eso en el intérprete:

```
>>> A=Nombre("Test")  
>>> A  
<Clase Nombre en 0x3000f10 Nombre: Test Clave: test>  
>>> A.nombre="Nilo"  
>>> A  
<Clase Nombre en 0x3000f10 Nombre: Nilo Clave: test>  
>>> A.clave="TST"  
>>> A  
<Clase Nombre en 0x3000f10 Nombre: Nilo Clave: TST>
```

Observe que después de la construcción del objeto, el nombre y la clave sean correctos, pues esa operación está garantizada por nuestra implementación de **__init__**. Mientras tanto, un acceso a **A.nombre** no actualizará el valor de la clave, que continúa con el valor configurado por el constructor. Lo mismo sucede con el atributo **clave** que puede ser alterado sin cualquier ligazón con el **nombre** en sí. En la programación orientada a objetos, una clase es responsable por administrar su estado interno y mantener la consistencia del objeto entre llamadas de métodos. Para garantizar que **nombre** y **clave** siempre sean actualizados correctamente, vamos hacer uso de un recurso de Python que utiliza métodos para realizar la atribución y la lectura de valores. Ese recurso es llamado de propiedad. El programa de la lista 10.16 también utilizará otro recurso, una convención especial de nomenclatura de nuestros métodos para esconder los atributos de la clase e impedir el acceso directo a ellos desde fuera de la clase.

► Lista 10.16 – Clase Nombre con propiedades (nombre.py)

```
from functools import total_ordering  
  
@total_ordering
```

```

class Nombre:
    def __init__(self, nombre):
        self.nombre = nombre ❶

    def __str__(self):
        return self.nombre

    def __repr__(self):
        return "<Clase {3} en 0x{0:x} Nombre: {1} Clave: {2}>".format(
            id(self), self.__nombre, self.__clave, type(self).__name__) ❷

    def __eq__(self, otro):
        return self.nombre == otro.nombre

    def __lt__(self, otro):
        return self.nombre < otro.nombre

    @property ❸
    def nombre(self):
        return self.__nombre ❹

    @nombre.setter ❺
    def nombre(self, valor):
        if valor == None or not valor.strip():
            raise ValueError("Nombre no puede ser nulo nem en blanco")
        self.__nombre = valor ❻
        self.__clave = Nombre.CreaClave(valor) ❼

    @staticmethod
    def CreaClave(nombre):
        return nombre.strip().lower()

```

El programa de la lista 10.16 utiliza otros decoradores como `@property`, en ❸, y `@nombre.setter`, en ❺. Esos decoradores modifican luego los métodos abajo de ellos, transformándolos en propiedades. El primer decorador, `@property`, transforma el método `nombre` en la propiedad `nombre`; de esa forma, siempre que escriba `objeto.nombre`, estará llamando ese método, que retorna `self.__nombre`. El segundo decorador, `@nombre.setter`, transforma el método `nombre` en la propiedad usada para alterar el valor de `__nombre`. De esa forma, cuando escriba `objeto.nombre = valor`, ese método será llamado para efectuar los cambios. Observe que copiamos la verificación de tipo y la creación de la clave del método `__init__` (en ❶) para el método marcado por `@nombre.setter` en ❺. Así, en ❶, llamamos al método de ❺ al escribir: `self.nombre = nombre`, una vez que `self.nombre` es ahora una propiedad. Otro detalle importante es que agregamos dos sublíneas (`__`) antes del nombre de los atributos `nombre` y `clave`, que quedaron como `__nombre` y `__clave`. De esa forma, `__nombre` y `__clave` quedan escondidos cuando son accedidos de fuera de la clase. Ese “esconder” es solo un detalle de la implementación de Python, que modifica el nombre de esos atributos de modo de volverlos inaccesibles (*name mangling*). Mientras tanto, sus nombres fueron transformados solo por la adición de `_`, y del nombre de la clase, haciendo que `__clave` se vuelva `_Nombre__clave`. Pero esa protección es suficiente para garantizar que `nombre` y `clave` estén

sincronizadas y que nuestra clase funcione de la manera esperada. En Python, preste mucha atención al utilizar nombres que empiezan con `_` o con `__`. Esos símbolos indican que esos atributos no deben ser accedidos, excepto por el código de la propia clase. No confundir los atributos protegidos, cuyo nombre comienza por `_`, con el nombre de los métodos mágicos especiales que ya vimos, que empiezan y terminan por `__`. En ❷, podemos ver que `__nombre` y `__clave` continúan accesibles dentro del código de la clase por `self.__nombre` y `self.__clave`.

Vamos a probar el nuevo código en el intérprete:

```
>>> from nombre import *
>>> A=Nombre("Nilo") ❶
>>> A
<Clase Nombre en 0x3140f10 Nombre: Nilo Clave: nilo>
>>> A.nombre="Nilo Menezes" ❷
>>> A
<Clase Nombre en 0x3140f10 Nombre: Nilo Menezes Clave: nilo menezes>
>>> A.__nombre ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Nombre' object has no attribute '__nombre'
>>> A.__clave ❹
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Nombre' object has no attribute '__clave'
>>> A.clave ❺
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Nombre' object has no attribute 'clave'
>>> A._Nombre__clave ❻
'nilo menezes'
```

En ❶, creamos el objeto A, como anteriormente. Vea que el `self.nombre` dentro del `__init__` llamó correctamente al método marcado por `@nombre.setter`, y que `clave` y `nombre` fueron correctamente configuradas. En ❷, alteramos el `nombre` y luego podemos ver que la clave fue actualizada al mismo tiempo. Observe que `A.nombre` es accesible desde fuera de la clase y que ahora forma parte de nuestro objeto como antes formaba el atributo `self.nombre`. ❸, ❹ y ❺ muestran que `__nombre`, `__clave` y `clave` son inaccesibles desde fuera de la clase. En ❻, mostramos el acceso a la clave, con el truco de usar el nombre completo después del ocultamiento de Python (*name mangling*): `_nomedaclase__atributo`, generando: `_Nombre__clave`. Considere ese tipo de acceso como una curiosidad y que, cuando un programador marca un atributo con `__`, está diciendo: no utilice ese atributo fuera de la clase, salvo si tiene certeza de lo que está haciendo.

El truco de los nombres que empiezan por `__` también funciona con métodos. Si usted quiere marcar

un método para ser utilizado solo dentro de la clase, adicione `__` antes de su nombre, del mismo modo que hicimos con nuestros atributos.

En todos los casos, nunca cree métodos o atributos que comiencen y terminen por `__`, ese tipo de construcción está reservado a los métodos mágicos (especiales) del lenguaje.

Puede crear atributos que pueden ser leídos definiendo el método de acceso solo con `@property`. En nuestro ejemplo, `@nombre.setter` es lo que permite alterar el nombre. Si no utilizásemos `@nombre.setter`, `A.nombre` sería accesible solo para lectura, y se generaría una excepción si intentásemos alterar su valor. Veamos cómo funciona eso en el programa de la lista 10.17.

► Lista 10.17 – Clave como propiedad solo para lectura (nombre.py)

```
from functools import total_ordering
@total_ordering
class Nombre:
    def __init__(self, nombre):
        self.nombre = nombre
    def __str__(self):
        return self.nombre
    def __repr__(self):
        return "<Clase {3} en 0x{0:x} Nombre: {1} Clave: {2}>".format(
            id(self), self.__nombre, self.__clave, type(self).__name__)
    def __eq__(self, otro):
        return self.nombre == otro.nombre
    def __lt__(self, otro):
        return self.nombre < otro.nombre
    @property
    def nombre(self):
        return self.__nombre
    @nombre.setter
    def nombre(self, valor):
        if valor == None or not valor.strip():
            raise ValueError("Nombre no puede ser nulo ni en blanco")
        self.__nombre = valor
        self.__clave = Nombre.CreaClave(valor)
    @property
    def clave(self):
        return self.__clave
    @staticmethod
    def CreaClave(nombre):
        return nombre.strip().lower()
```

Vea que en la lista 10.17, marcamos solo el método con `@property` y no creamos un método con `@clave.setter` para procesar las modificaciones. En el intérprete, veamos qué sucede si tratamos de alterar clave:

```
>>>from nombre import *
>>> A=Nombre("Nilo")
>>> A.clave
'nilo'
>>> A.clave="nilo menezes"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>> A.nombre="Nilo Menezes"
>>> A.clave
'nilo menezes'
```

10.6 Revisando la agenda

En el capítulo 9 desarrollamos una agenda de nombres y teléfonos. Ahora que conocemos las clases, podemos adaptar el código de la agenda de modo de mejorar su funcionamiento y utilizar algunos conceptos de programación orientada a objetos.

Vamos a revisar la agenda de modo de definir sus estructuras como objetos y modelar la aplicación de la agenda en una clase separada. En las secciones anteriores, definimos las clases **Nombre** y **ListaÚnica**. Esas clases serán utilizadas en el modelo de datos de nuestra nueva agenda.

Podemos visualizar nuestra agenda como una lista de nombres y teléfonos. La agenda original del capítulo 9 soportaba solo un teléfono por nombre, pero modificamos eso en el ejercicio 9.28. Por lo tanto, nuestra nueva agenda deberá admitir varios teléfonos, y cada teléfono tendrá un tipo específico (celular, trabajo, fax etc.).

Vamos a empezar por la clase que va a administrar los tipos de teléfono. Veamos el código en la lista 10.18.

► Lista 10.18 – La clase TipoTeléfono

```
from functools import total_ordering
@total_ordering
class TipoTeléfono:
    def __init__(self, tipo):
        self.tipo = tipo
    def __str__(self):
        return "({0})".format(self.tipo)
    def __eq__(self, otro):
        if otro is None:
```

```

        return False
    return self.tipo == otro.tipo
def __lt__(self, otro):
    return self.tipo < otro.tipo

```

El programa de la lista 10.18 implementa nuestro tipo de teléfono. Vea que implementamos el método `__str__` para exhibir el nombre del tipo entre paréntesis, y el método `__eq__` y `__lt__` para activar la comparación por tipo. En ese ejemplo no usamos propiedades, pues en la clase `TipoTeléfono` no estamos haciendo ninguna verificación. De esa forma, `self.tipo` ofrecerá la misma forma de acceso, ya sea como atributo o propiedad. La gran ventaja de utilizar propiedades es que, si es necesario, más tarde podemos transformar `self.tipo` en una propiedad, sin alterar el código que utiliza la clase, tal como hicimos con el nombre en la clase `Nombre`. Observe que en el método `__eq__`, incluimos una verificación si la otra parte es `None`. Esa verificación es necesaria porque nuestra futura agenda utilizará `None` cuando no sepamos el tipo de un teléfono. Así, `self.tipo == otro.tipo` fallaría, pues `None` no tiene un atributo llamado `tipo`. El test de si otro es `None` fue hecho con el operador `is` de Python. Esto es importante porque si utilizamos `==`, estaremos llamando recursivamente el método `__eq__` que es llamado por el operador de comparación (`==`).

La clase `Teléfono` será implementada por el programa de la lista 10.19.

► Lista 10.19 – La clase `Teléfono`

```

class Teléfono:
    def __init__(self, número, tipo=None):
        self.número = número
        self.tipo = tipo
    def __str__(self):
        if self.tipo is not None:
            tipo = self.tipo
        else:
            tipo = ""
        return "{0} {1}".format(self.número, tipo)
    def __eq__(self, otro):
        return self.número == otro.número and (
            (self.tipo == otro.tipo) or (
                self.tipo is None or otro.tipo is None))
    @property
    def número(self):
        return self.__número
    @número.setter
    def número(self, valor):
        if valor is None or not valor.strip():

```

```
        raise ValueError("Número no puede ser None o en blanco")
    self.__numero = valor
```

En la lista 10.19, hacemos el mismo tipo de verificación de tipo que hicimos en la clase **Nombre**. De esa forma, un objeto de la clase **Teléfono** no acepta números vacíos. En el método `__eq__`, implementamos la verificación de modo de ignorar un **Teléfono** sin **TipoTeléfono**, o sea, un teléfono donde el tipo es **None**. De esa forma, si comparamos dos objetos de **Teléfono** y uno o ambos tienen el tipo **None**, el resultado de la comparación será decidido solo si el número es idéntico. Si las dos instancias de **Teléfono** tienen un tipo válido (diferente de **None**), entonces el tipo formará parte de la comparación, siendo iguales solo si los números y los tipos son iguales. Esas decisiones fueron tomadas para la implementación de la agenda y son decisiones de proyecto. Eso significa que para otras aplicaciones, esas decisiones podrían ser diferentes. El proceso quedará más claro cuando leamos el programa completo de la agenda.

Ahora que tenemos las clases **Nombre**, **Teléfono** y **TipoTeléfono**, podemos pasar a pensar en cómo organizar los objetos de esas clases. Podemos ver nuestra agenda como una gran lista, donde cada elemento es un compuesto de **Nombre** con una lista de objetos del tipo **Teléfono**. Vamos a llamar cada entrada en nuestra agenda de **DatoAgenda** e implementar una clase para agrupar instancias de las dos clases. Vea el programa de la lista 10.20 con la clase **DatoAgenda**.

► Lista 10.20 – Clase **DatoAgenda**

```
import listaunica
class Telefonos(ListaÚnica):
    def __init__(self):
        super().__init__(Teléfono)
class DatoAgenda:
    def __init__(self, nombre):
        self.nombre = nombre
        self.teléfonos = Telefonos()
    @property
    def nombre(self):
        return self.__nombre
    @nombre.setter
    def nombre(self, valor):
        if type(valor)!=Nombre:
            raise TypeError("nombre debe ser una instancia de la clase Nombre")
        self.__nombre = valor
    def investigaciónTeléfono(self, teléfono):
        posición = self.teléfonos.investigación(Teléfono(teléfono))
        if posición == -1:
            return None
        else:
```



```
return self.teléfonos[posición]
```

Como cada objeto de **DatoAgenda** puede contener varios teléfonos, creamos una clase **Teléfonos** que hereda de la clase **ListaÚnica** su comportamiento. De esa forma **self.nombre** de **DatoAgenda** es una instancia de la clase **Nombre**, y **self.teléfonos** una instancia de **Teléfonos**. Vea que adicionamos una propiedad **nombre** para facilitar el acceso al objeto de **self.__nombre** y hacer las verificaciones de tipo necesarias. Adicionamos también un método **investigaciónTeléfono** que transforma una cadena de caracteres en un objeto de la clase **Teléfono**, sin tipo. Ese objeto es entonces utilizado por el método **investigación**, surgido de la implementación original de **ListaÚnica** que retorna la posición del objeto en la lista, o **-1**, en caso que no sea encontrado. Observe que el método **investigaciónTeléfono** devuelve una instancia de **Teléfono** si ella es encontrada en la lista o **None**, en caso contrario.

Veamos ahora la clase **Agenda** en la lista 10.21.

► Lista 10.21 – Lista parcial del programa de la agenda

```
class TiposTeléfono(ListaÚnica):  
    def __init__(self):  
        super().__init__(TipoTeléfono)  
class Agenda(ListaÚnica):  
    def __init__(self):  
        super().__init__(DatoAgenda)  
        self.tiposTeléfono = TiposTeléfono()  
    def adicionaTipo(self, tipo):  
        self.tiposTeléfono.adiciona(TipoTeléfono(tipo))  
    def investigaciónNombre(self, nombre):  
        if type(nombre) == str:  
            nombre = Nombre(nombre)  
        for datos in self.lista:  
            if datos.nombre == nombre:  
                return datos  
        else:  
            return None  
    def ordena(self):  
        super().ordena(lambda dato: str(dato.nombre))
```

El programa de la lista 10.21 es una lista parcial, solo para explicar la clase **Agenda**. El programa completo necesita importar las definiciones de clase hechas anteriormente. Observe que la clase **Agenda** es una subclase de **ListaÚnica** configurada para aceptar solamente objetos del tipo **DatoAgenda**. Vea que definimos una clase **TiposTeléfono**, también heredando de **ListaÚnica**, para mantener la lista de tipos de teléfono. En nuestra agenda, los tipos de teléfono son preconfigurados en la clase **Agenda**. Para facilitar el trabajo de inclusión de nuevos tipos, incluimos el método **adicionaTipo** que prepara una cadena de caracteres, transformándola en objeto de **TipoTeléfono** e

incluyéndola en la lista de tipos válidos. Otro método a considerar es `investigaciónNombre`, que recibe el nombre a investigar como objeto de la clase `Nombre` o como cadena de caracteres. Vea que utilizamos la verificación del tipo del parámetro nombre para transformarlo en objeto de `Nombre`, en caso necesario, dejando que nuestra clase funcione con cadenas de caracteres o con objetos del tipo `Nombre`. La función entonces investiga en la lista interna de datos y retorna el objeto, en caso que sea encontrado. Es importante notar la diferencia de esa función de investigación y otras funciones de investigación que definimos anteriormente. En `investigaciónNombre`, el objeto es retornado o `None`, en caso que no sea encontrado. Del mismo modo que vimos el problema de referencias en listas en el capítulo 6, vamos a utilizar las referencias retornadas por `investigaciónNombre` para editar los valores de la instancia de `DatoAgenda` directamente, sin necesidad de reintroducirlas en la lista. Eso quedará más claro cuando analicemos el programa de la agenda completo. Para finalizar, el método `ordena` crea una función para extraer la clave de ordenamiento de `DatoAgenda`; en ese caso, el `nombre`.

Antes de pasar al programa de la agenda, vamos a construir una clase `Menú` para exhibir el menú principal. Vea el programa de la lista 10.22.

► Lista 10.22 – Lista parcial de la agenda: clase `Menú`

```
class Menú:
```

```
    def __init__(self):
        self.opciones = ["Salir", None]
    def adicionaopción(self, nombre, función):
        self.opciones.append([nombre, función])
    def exhibe(self):
        print("====")
        print("Menú")
        print("====\n")
        for i, opción in enumerate(self.opciones):
            print("[{0}] - {1}".format(i, opción[0]))
        print()
    def ejecute(self):
        while True:
            self.exhibe()
            elija = valida_franja_entero("Elija una opción: ",
                                         0, len(self.opciones)-1)
            if elija == 0:
                break
            self.opciones[elija][1]()
```

Nuestro menú es bien simple, adicionando la opción para "Salir" como valor estándar. El método `exhibe` muestra el menú en la pantalla, recorriendo la lista de opciones. El método `ejecute` muestra continuamente el menú, pidiendo una elección y llamando al método correspondiente. Veamos

cómo utilizar la clase `Menú` en la inicialización de la clase `AppAgenda` en la lista 10.23:

► Lista 10.23 – Lista completa de la nueva agenda

```
import sys
import pickle
from functools import total_ordering

def nulo_o_vacio(texto):
    return texto == None or not texto.strip()

def valida_franja_entero(pregunta, inicio, fin, estándar=None):
    while True:
        try:
            entrada = input(pregunta)
            if nulo_o_vacio(entrada) and estándar != None:
                entrada = estándar
            valor = int(entrada)
            if inicio <= valor <= fin:
                return(valor)
        except ValueError:
            print("Valor inválido, favor digitar entre %d y %d" %
                  (inicio, fin))

def valida_franja_entero_o_blanco(pregunta, inicio, fin):
    while True:
        try:
            entrada = input(pregunta)
            if nulo_o_vacio(entrada):
                return None
            valor = int(entrada)
            if inicio <= valor <= fin:
                return(valor)
        except ValueError:
            print("Valor inválido, favor digitar entre %d y %d" %
                  (inicio, fin))

class ListaÚnica:
    def __init__(self, elem_class):
        self.lista = []
        self.elem_class = elem_class
    def __len__(self):
        return len(self.lista)
    def __iter__(self):
```

```

    return iter(self.lista)
def __getitem__(self, p):
    return self.lista[p]
def indiceVálido(self, i):
    return i>=0 and i<len(self.lista)
def adiciona(self, elem):
    if self.investigación(elem) == -1:
        self.lista.append(elem)
def remove(self, elem):
    self.lista.remove(elem)
def investigación(self, elem):
    self.verifica_tipo(elem)
    try:
        return self.lista.index(elem)
    except ValueError:
        return -1
def verifica_tipo(self, elem):
    if type(elem)!=self.elem_class:
        raise TypeError("Tipo inválido")
def ordena(self, clave=None):
    self.lista.sort(key=clave)

```

@total_ordering

class Nombre:

```

    def __init__(self, nombre):
        self.nombre = nombre
    def __str__(self):
        return self.nombre
    def __repr__(self):
        return "<Clase {3} en 0x{0:x} Nombre: {1} Clave: {2}>".format(
            id(self), self.__nombre, self.__clave,
            type(self).__name__)
    def __eq__(self, otro):
        return self.nombre == otro.nombre
    def __lt__(self, otro):
        return self.nombre < otro.nombre
@property
def nombre(self):
    return self.__nombre

```

```

@nombre.setter
def nombre(self, valor):
    if nulo_o_vacio(valor):
        raise ValueError("Nombre no puede ser nulo ni en blanco")
    self.__nombre = valor
    self.__clave = Nombre.CreaClave(valor)

@property
def clave(self):
    return self.__clave

@staticmethod
def CreaClave(nombre):
    return nombre.strip().lower()

```

```

@total_ordering
class TipoTeléfono:
    def __init__(self, tipo):
        self.tipo = tipo
    def __str__(self):
        return "({0})".format(self.tipo)
    def __eq__(self, otro):
        if otro is None:
            return False
        return self.tipo == otro.tipo
    def __lt__(self, otro):
        return self.tipo < otro.tipo

```

```

class Teléfono:
    def __init__(self, número, tipo=None):
        self.número = número
        self.tipo = tipo
    def __str__(self):
        if self.tipo!=None:
            tipo = self.tipo
        else:
            tipo = ""
        return "{0} {1}".format(self.número, tipo)
    def __eq__(self, otro):
        return self.número == otro.número and (
            (self.tipo == otro.tipo) or (
                self.tipo == None or otro.tipo == None))

```

```

@property
def número(self):
    return self.__número
@número.setter
def número(self, valor):
    if nulo_o_vacio(valor):
        raise ValueError("Número no puede ser None o en blanco")
    self.__número = valor
class Teléfonos(ListaÚnica):
    def __init__(self):
        super().__init__(Teléfono)
class TiposTeléfono(ListaÚnica):
    def __init__(self):
        super().__init__(TipoTeléfono)
class DatoAgenda:
    def __init__(self, nombre):
        self.nombre = nombre
        self.teléfonos = Teléfonos()
@property
def nombre(self):
    return self.__nombre
@nombre.setter
def nombre(self, valor):
    if type(valor)!=Nombre:
        raise TypeError("nombre debe ser una instancia de la clase Nombre")
    self.__nombre = valor
def investigaciónTeléfono(self, teléfono):
    posición = self.teléfonos.investigación(Teléfono(teléfono))
    if posición == -1:
        return None
    else:
        return self.teléfonos[posición]
class Agenda(ListaÚnica):
    def __init__(self):
        super().__init__(DatoAgenda)
        self.tiposTeléfono = TiposTeléfono()
    def adicionaTipo(self, tipo):
        self.tiposTeléfono.adiciona(TipoTeléfono(tipo))
    def investigaciónNombre(self, nombre):

```

```

    if type(nombre) == str:
        nombre = Nombre(nombre)
    for datos in self.lista:
        if datos.nombre == nombre:
            return datos
    else:
        return None
def ordena(self):
    super().ordena(lambda dato: str(dato.nombre))
class Menú:
    def __init__(self):
        self.opciones = [["Salir", None]]
    def adicionaopción(self, nombre, función):
        self.opciones.append([nombre, función])
    def exhibe(self):
        print("====")
        print("Menú")
        print("====\n")
        for i, opción in enumerate(self.opciones):
            print("[{0}] - {1}".format(i, opción[0]))
        print()
    def ejecute(self):
        while True:
            self.exhibe()
            elija = valida_franja_entero("Elija una opción: ",
                                         0, len(self.opciones)-1)
            if elija == 0:
                break
            self.opciones[elija][1]()
class AppAgenda:
    @staticmethod
    def pide_nombre():
        return(input("Nombre: "))
    @staticmethod
    def pide_teléfono():
        return(input("Teléfono: "))
    @staticmethod
    def muestra_datos(datos):

```

```

print("Nombre: %s" % datos.nombre)
for teléfono in datos.teléfonos:
    print("Teléfono: %s" % teléfono)
print()
@staticmethod
def muestra_datos_teléfono(datos):
    print("Nombre: %s" % datos.nombre)
    for i, teléfono in enumerate(datos.teléfonos):
        print("{0} - Teléfono: {1}".format(i, teléfono))
    print()
@staticmethod
def pide_nombre_archivo():
    return(input("Nombre del archivo: "))
def __init__(self):
    self.agenda = Agenda()
    self.agenda.adicionaTipo("Celular")
    self.agenda.adicionaTipo("Residencia")
    self.agenda.adicionaTipo("Trabajo")
    self.agenda.adicionaTipo("Fax")
    self.menú = Menú()
    self.menú.adicionaopción("Nuevo", self.nuevo)
    self.menú.adicionaopción("Alterar", self.altera)
    self.menú.adicionaopción("Borra", self.borra)
    self.menú.adicionaopción("Lista", self.lista)
    self.menú.adicionaopción("Graba", self.graba)
    self.menú.adicionaopción("Lee", self.lee)
    self.menú.adicionaopción("Ordena", self.ordena)
    self.ultimo_nombre = None
def pide_tipo_teléfono(self, estándar=None):
    for i,tipo in enumerate(self.agenda.tiposTeléfono):
        print(" {0} - {1} ".format(i,tipo),end=None)
    t = valida_franja_entero("Tipo: ",0, len(self.agenda.tiposTeléfono)-1,
estándar)
    return self.agenda.tiposTeléfono[t]
def investigación(self, nombre):
    dato = self.agenda.investigaciónNombre(nombre)
    return dato
def nuevo(self):
    nuevo = AppAgenda.pide_nombre()

```



```

if nulo_o_vacio(nuevo):
    return
nombre = Nombre(nuevo)
if self.investigación(nombre) != None:
    print("¡Nombre ya existe!")
    return
registro = DatoAgenda(nombre)
self.menú_teléfonos(registro)
self.agenda.adiciona(registro)
def borra(self):
    if len(self.agenda)==0:
        print("Agenda vacía, nada a borrar")
    nombre = AppAgenda.pide_nombre()
    if(nulo_o_vacio(nombre)):
        return
    p = self.investigación(nombre)
    if p != None:
        self.agenda.remove(p)
        print("Borrado. La agenda ahora tiene solo: %d registros" %
len(self.agenda))
    else:
        print("Nombre no encontrado.")
def altera(self):
    if len(self.agenda)==0:
        print("Agenda vacía, nada a alterar")
    nombre = AppAgenda.pide_nombre()
    if(nulo_o_vacio(nombre)):
        return
    p = self.investigación(nombre)
    if p != None:
        print("\\nEncontrado:\\n")
        AppAgenda.muestra_datos(p)
        print("Digite enter en caso que no quiera alterar el nombre")
        nuevo = AppAgenda.pide_nombre()
        if not nulo_o_vacio(nuevo):
            p.nombre = Nombre(nuevo)
            self.menú_teléfonos(p)
        else:
            print("¡Nombre no encontrado!")

```

```

def menú_teléfonos(self, datos):
    while True:
        print("\nEditando teléfonos\n")
        AppAgenda.muestra_datos_teléfono(datos)
        if(len(datos.teléfonos)>0):
            print("\n[A] - alterar\n[D] - borrar\n", end="")
            print("[N] - nuevo\n[S] - salir\n")
            operación = input("Elija una operación: ")
            operación = operación.lower()
            if operación not in ["a","d","n", "s"]:
                print("Operación inválida. Digite A, D, N o S")
                continue
            if operación == 'a' and len(datos.teléfonos)>0:
                self.altera_teléfonos(datos)
            elif operación == 'd' and len(datos.teléfonos)>0:
                self.borra_teléfono(datos)
            elif operación == 'n':
                self.nuevo_teléfono(datos)
            elif operación == "s":
                break

def nuevo_teléfono(self, datos):
    teléfono = AppAgenda.pide_teléfono()
    if nulo_o_vacío(teléfono):
        return
    if datos.investigaciónTeléfono(teléfono) != None:
        print("Teléfono ya existe")
    tipo = self.pide_tipo_teléfono()
    datos.teléfonos.adiciona(Teléfono(teléfono, tipo))

def borra_teléfono(self, datos):
    t = valida_franja_entero_o_blanco(
        "Digite la posición del número a borrar, enter para salir: ",
        0, len(datos.teléfonos)-1)
    if t == None:
        return
    datos.teléfonos.remove(datos.teléfonos[t])

def altera_teléfonos(self, datos):
    t = valida_franja_entero_o_blanco(
        "Digite la posición del número a alterar, enter para salir: ",
        0, len(datos.teléfonos)-1)

```

```

if t == None:
    return
telefono = datos.telefonos[t]
print("Teléfono: %s" % telefono)
print("Digite enter en caso que no quiera alterar el número")
nuevoteléfono = AppAgenda.pide_telefono()
if not nulo_o_vacio(nuevoteléfono):
    telefono.número = nuevoteléfono
print("Digite enter en caso que no quiera alterar el tipo")
telefono.tipo = self.pide_tipo_telefono(
    self.agenda.tiposTeléfono.investigación(telefono.tipo))
def lista(self):
    print("\\nAgenda")
    print("-"*60)
    for e in self.agenda:
        AppAgenda.muestra_datos(e)
    print("-"*60)
def lee(self, nombre_archivo=None):
    if nombre_archivo == None:
        nombre_archivo = AppAgenda.pide_nombre_archivo()
    if nulo_o_vacio(nombre_archivo):
        return
    with open(nombre_archivo, "rb") as archivo:
        self.agenda = pickle.load(archivo)
        self.ultimo_nombre = nombre_archivo
def ordena(self):
    self.agenda.ordena()
    print("\\nAgenda ordenada\\n")
def graba(self):
    if self.ultimo_nombre != None:
        print("Último nombre utilizado fue '%s'" % self.ultimo_nombre)
        print("Digite enter en caso que quiera utilizar el mismo nombre")
        nombre_archivo = AppAgenda.pide_nombre_archivo()
    if nulo_o_vacio(nombre_archivo):
        if self.ultimo_nombre != None:
            nombre_archivo = self.ultimo_nombre
        else:
            return

```

```

        with open(nombre_archivo, "wb") as archivo:
            pickle.dump(self.agenda, archivo)

    def ejecute(self):
        self.menú.ejecute()

if __name__ == "__main__":
    app = AppAgenda()
    if len(sys.argv) > 1:
        app.lee(sys.argv[1])
    app.ejecute()

```

La lista 10.23 contiene el programa completo, donde todas las clases necesarias fueron escritas en un solo archivo para facilitar la lectura. La clase **AppAgenda** es nuestra aplicación de la agenda en sí. Observe la implementación de `__init__`, donde creamos una instancia de **Agenda** para contener nuestros datos e inicializamos los tipos de teléfono que queremos trabajar. También creamos una instancia de **Menú** y adicionamos las opciones. Observe que pasamos el nombre de la opción y el método correspondiente a cada elección. El atributo `self.ultimo_nombre` es usado para guardar el nombre usado en la última lectura del archivo.

En la nueva agenda, note que dividimos el tratamiento de nombres y teléfonos. Como podemos tener varios teléfonos, aparece otro menú con opciones para la administración de teléfonos. Puede ver los detalles en la implementación del método `menú_teléfonos`. En nuestra agenda adoptamos el siguiente comportamiento cuando un valor no cambia, digitamos solo **Enter**. En todas las opciones, usted debe observar el tratamiento de entradas en blanco, incluso creamos una función de soporte para eso: `nulo_o_vacio`.

En todos los métodos de edición puede ver que utilizamos solo los métodos provistos por la clase **ListaÚnica**. Vea también que las investigaciones retornan los objetos y no solo la posición de ellos en la lista.

Los métodos `graba` y `lee` fueron modificados para utilizar el módulo **pickle** de Python. Como nuestra agenda ahora tiene varios teléfonos, cada uno con un tipo, la utilización de archivos simple se volvería muy trabajosa. El módulo **pickle** provee métodos que graban un objeto o una lista de objetos en el disco. En el método `graba`, utilizamos la función `pickle.dump` que graba en el archivo el objeto agenda entero. En el método `lee`, utilizamos `pickle.load` para recrear nuestra agenda a partir del archivo en el disco. La utilización del **pickle** facilita mucho la tarea de serializar los datos en un archivo, o sea, de representar los objetos de modo que puedan ser reconstruidos en caso necesario (proceso de serialización).

Como última modificación, adicionamos la posibilidad de pasar el nombre del archivo de la agenda como parámetro. En caso que el nombre sea pasado, el archivo es leído antes de presentar el menú. Observe también que el método `ejecute` tiene un loop infinito que rueda hasta salir del menú principal.

Banco de datos

Prácticamente todo programa necesita leer o almacenar datos. Para una cantidad pequeña de información, archivos simples resuelven el problema; pero una vez que los datos precisan ser actualizados, empiezan a aparecer problemas de mantenimiento y dificultad de acceso.

Después de algún tiempo, el programador comienza a ver que las operaciones realizadas con archivos siguen algunos estándares, como por ejemplo: introducir, alterar, borrar e investigar. También comienza a observar que las consultas pueden ser hechas con criterios diferentes, y que a medida que el archivo crece, las operaciones se vuelven más lentas. Todos esos tipos de problemas fueron identificados y resueltos hace bastante tiempo, con el uso de sistemas gestores de base de datos.

Los sistemas gestores de base de datos fueron desarrollados de modo de organizar y facilitar el acceso a grandes masas de información. Mientras tanto, para usar un banco de datos, necesitamos saber cómo están organizados. En este capítulo, abordaremos los conceptos básicos de banco de datos, así como la utilización del lenguaje SQL y el acceso vía lenguaje de programación, en este caso, Python.

Las listas del capítulo 11 son mucho mayores que las del resto del libro. Se recomienda leer este capítulo cerca de una computadora y en algunos casos con las listas impresas en papel, o fácilmente accesibles en el sitio del libro.

11.1 Conceptos básicos

Para comenzar a utilizar los términos del banco de datos, es importante entender cómo funcionaban las cosas antes de usar computadoras para controlar el registro de informaciones.

No mucho tiempo atrás, las personas usaban su propia memoria para almacenar informaciones importantes, como los números de teléfonos de amigos próximos y hasta el propio número de teléfono. Hoy, con la proliferación de celulares, prácticamente perdimos esa capacidad, por falta de uso. Sin embargo, para entender cómo funciona un banco de datos, necesitamos comprender por qué fueron creados, qué problemas resolvieron y cómo eran las cosas antes de ellos.

Imagine que usted es una persona extremadamente popular y que necesita controlar más de 100 contactos (nombre y teléfono) de los amigos más próximos. Imagine esa situación en un mundo sin computadoras. ¿Qué haría para controlar todos los datos?

Probablemente escribiría todos los nombres y teléfonos de sus nuevos contactos en hojas de papel. Usando un cuaderno, se podría anotar fácilmente un nombre debajo del otro, con el número de teléfono anotado a la derecha de la hoja.

Tabla 11.1 – Ejemplo de nombres anotados en una hoja de papel

Nombre	Teléfono
Juan	98901-0109
André	98902-8900
María	97891-3321

El problema con las anotaciones en papel es que no conseguimos alterar los datos allí escritos, salvo si escribimos a lápiz, pero el resultado nunca es muy agradable. Otro problema es que no conocemos nuevas personas en orden alfabético, lo que resulta en una lista de nombres y teléfonos desordenada. El papel tampoco ayuda cuando tenemos más amigos con nombres que empiezan por una letra que por otra; o cuando nuestros amigos tienen varios nombres, como Juan Carlos, y ¡usted nunca recuerda si lo registró como Juan o como Carlos!

Ahora bien, imagine que estamos controlando no solo nuestros amigos, sino también contactos comerciales. En el caso de los amigos, solo nombre y teléfono bastan para restablecer el contacto. En el caso de un contacto comercial, la empresa donde la persona trabaja y la función que desempeña también son importantes. Con el tiempo necesitaríamos varios cuadernos o carpetas, uno para cada contacto, organizados en armarios, tal vez un cajón para cada tipo de registro... Antes que las computadoras se volvieran populares, las cosas eran organizadas de esa forma, y en muchos lugares aún hoy son así.

Una vez que los problemas que pueden ser resueltos con un banco de datos fueron presentados, ya podemos aprender algunos conceptos importantísimos para continuar el estudio, tales como: campos, registros, tablas y tipos de datos.

Campos son la menor unidad de información en un banco de datos. Si hacemos una comparación con nuestra agenda en el papel, nombre y teléfono serían dos campos. El campo nombre almacenaría el nombre de cada contacto; y el campo teléfono, el número de teléfono, respectivamente.

Cada línea de nuestra agenda sería llamada registro. Un registro está formado por un conjunto conocido de campos. En nuestro ejemplo, cada persona en la agenda, con su nombre y teléfono, formaría un registro.

Podemos pensar las tablas del banco de datos como la unidad de almacenamiento de registros del mismo tipo. Imagine una entrada de la agenda telefónica, donde cada registro, conteniendo nombre y teléfono, es almacenado. El conjunto de registros del mismo tipo es organizado en tablas, en ese caso, en la tabla agenda o lista telefónica.

Tabla

The diagram illustrates a database table structure. At the top, a bracket labeled 'Tabla' spans the entire table. Below it, two brackets labeled 'Campo Nombre' and 'Campo Teléfono' are positioned above the first two columns, respectively. The table itself has a header row with 'Nombre' and 'Teléfono' in grey cells. Below the header are three data rows. A bracket on the left labeled 'Registros' spans these three rows. The data rows contain the following values:

Nombre	Teléfono
Maria	97891-3321
André	98902-8900
João	98901-0109

Figura 11.1 – Campos, registros y tabla.

Los conceptos de campo, registro y tabla son fundamentales para el entendimiento del resto de texto, ver figura 11.1. No dude en releer esa sección, en caso que alguno de esos conceptos aun no esté claro para usted. Los mismos serán representados en las secciones siguientes, cuando serán aplicados en la ejemplificación de un banco de datos.

11.2 SQL

Structured Query Language (SQL – Lenguaje de Consulta estructurada) es el lenguaje usado para crear bancos de datos, generar consultas, manipular (introducir, actualizar, alterar y borrar) registros y, principalmente, realizar consultas. Es un lenguaje de programación especializado en la manipulación de datos, basado en el álgebra relacional y en el modelo relacional creado por Edgar F. Codd (http://es.wikipedia.org/wiki/Edgar_Frank_Codd).

En este capítulo veremos cómo escribir comandos SQL para el banco SQLite, que viene preinstalado con el intérprete de Python y que es fácilmente accesible desde un programa. El lenguaje SQL está definido por varios estándares, como SQL-92, pero cada banco de datos introduce modificaciones y adiciones al estándar, aunque el funcionamiento básico se mantiene igual. En este capítulo, veremos exclusivamente los comandos SQL en el formato aceptado por el banco SQLite.

11.3 Python & SQLite

El SQLite es un sistema gestor de base liviano y completo, muy utilizado y presente incluso en teléfonos celulares. Una de sus principales características es no necesitar de un servidor dedicado, siendo capaz de funcionar a partir de su programa. En esta sección veremos los comandos más importantes y las etapas necesarias para utilizar el SQLite. Veamos un programa Python que crea un banco de datos, una tabla y un registro en la lista 11.1.

► Lista 11.1 Ejemplo de uso del SQLite en Python

```

import sqlite3 ❶
conexión = sqlite3.connect("agenda.db") ❷
cursor = conexión.cursor() ❸
cursor.execute('''
    create table agenda(
        nombre text,
        teléfono text)
    ''') ❹
cursor.execute('''
    insert into agenda (nombre, teléfono)
    values(?, ?)
    ''', ("Nilo", "7788-1432")) ❺
conexión.commit() ❻
cursor.close() ❼
conexión.close() ❽

```

La primera cosa a hacer es informar que utilizaremos un banco SQLite. Esto está hecho en ❶. Después del **import**, varias funciones y objetos que acceden al banco de datos se vuelven disponibles para su programa. Antes de continuar, vamos a crear el banco de datos en ❷. La conexión con el banco de datos se asemeja a la manipulación de un archivo, es una operación análoga a abrir un archivo. El nombre del banco de datos que estamos creando será grabado en el archivo **agenda.db**. La extensión **.db** es solo una convención, pero es recomendable diferenciar el nombre del archivo de un archivo normal, principalmente porque todos sus datos serán guardados en ese archivo. La gran ventaja de un banco de datos es que el registro de informaciones y todo el mantenimiento de los datos son hechos automáticamente con comandos SQL.

En ❸, creamos un cursor. Los cursores son objetos utilizados para enviar comandos y recibir resultados del banco de datos. Un cursor es creado para una conexión, llamándose el método **cursor()**. Una vez que obtuvimos un cursor, podemos enviar comandos al banco de datos. El primero de ellos es crear una tabla para guardar nombres y teléfonos. Vamos a llamarla agenda:

```

create table agenda(nombre text, teléfono text)

```

El comando SQL usado para crear una tabla es **create table**. Ese comando necesita el nombre de la tabla a crear; en este ejemplo **agenda**, y una lista de campos entre paréntesis. **Nombre** y **teléfono** son nuestros campos y **text** es el tipo. Aunque en Python no precisemos declarar el tipo de una variable, la mayoría de los bancos de datos exige un tipo para cada campo. En el caso del SQLite, el tipo no es exigido, pero vamos a continuar usándolo para que usted no tenga problemas con otros bancos, y para que la noción de tipo empiece a tener sentido. Un campo del tipo **text** puede almacenar datos como una cadena de caracteres de Python.

En ❹, utilizamos el método **execute** de nuestro cursor para enviar el comando al banco de datos. Observe que escribimos el comando en varias líneas, usando comillas triples de Python. El lenguaje SQL no exige ese formateo, aunque el mismo deja el comando más claro y simple de entender. Usted podría haber escrito todo en una sola línea y aún utilizar una cadena de caracteres simple de

Python.

Con la tabla creada, podemos empezar a introducir nuestros datos. Veamos el comando SQL usado para introducir un registro:

```
insert into agenda (nombre, teléfono) values (?, ?)
```

El comando **insert** necesita el nombre de la tabla, donde iremos a introducir los datos, y también el nombre de los campos y sus respectivos valores. **into** hace parte del comando **insert** y es escrito antes del nombre de la tabla. Los nombres de los campos se escriben luego, a continuación, separados por coma, y esta vez no necesitamos informar más el tipo de los campos, solo la lista de nombres. Los valores que vamos a introducir en la tabla son especificados también entre paréntesis, pero en la segunda parte del comando **insert** que comienza después de la palabra **values**. En nuestro ejemplo, la posición de cada valor fue marcada con signos de interrogación, uno para cada campo. El orden de los valores es el mismo de los campos; luego, la primera interrogación se refiere al campo **nombre**; la segunda, al campo **teléfono**. El lenguaje SQL permite que escribamos los valores directamente en el comando, como una gran cadena de caracteres; pero hoy en día, ese tipo de sintaxis no es recomendable por ser insegura y fácilmente utilizable para generar un ataque de seguridad de datos llamado SQLInjection (https://es.wikipedia.org/wiki/Inyecci%C3%B3n_SQL). Usted no necesita preocuparse de eso ahora, sobre todo porque al utilizar los signos de interrogación estamos utilizando parámetros que evitan ese tipo de problemas. Podemos entender los signos de interrogación como un equivalente de las máscaras de cadena de caracteres de Python, pero que utilizaremos con comandos SQL.

En ❸, utilizamos el método **execute** para ejecutar el comando **insert**, pero, esta vez, pasamos los datos después del comando. En el ejemplo, "Nilo" y "7788-1432" sustituirán el primer y el segundo signo de interrogación cuando el comando sea ejecutado. Es importante notar que los dos valores fueron pasados como una tupla.

Una vez que el comando es ejecutado, los datos son enviados al banco de datos, pero aún no están grabados definitivamente. Eso sucede porque estamos usando una transacción. Las transacciones serán presentadas con más detalle en otra sección; por ahora, considere el comando **commit** en ❹ como parte de las operaciones necesarias para modificar el banco de datos.

Antes de terminar el programa, cerramos (**close**) el cursor y la conexión con el banco de datos respectivamente, en ❺ y ❻. Veremos más adelante cómo usar la sentencia **with** de Python para facilitar esas operaciones.

Ejecute el programa y verifique si el archivo **agenda.db** fue creado. Si usted ejecuta el programa una segunda vez, será generado un error con el mensaje:

```
Traceback (most recent call last):
```

```
File "criatabla.py", line 9, in <module>
    '''
```

```
sqlite3.OperationalError: table agenda already exists
```

Este error sucede porque la tabla **agenda** ya existe. Si usted necesita ejecutar el programa nuevamente, borre el archivo **agenda.db**. Recuerde que todos los datos están en ese archivo, y al borrarlo todo se pierde. Puede borrar ese archivo siempre que quiera reinicializar el banco de datos.

Veamos ahora cómo leer los datos que grabamos en el banco de datos; vamos a hacer una consulta (*query*). El programa de la lista 11.2 realiza la consulta y muestra los resultados en la pantalla.

► Lista 11.2 – Consulta

```
import sqlite3
conexión = sqlite3.connect("agenda.db")
cursor = conexión.cursor()
cursor.execute("select * from agenda") ❶
resultado=cursor.fetchone() ❷
print("Nombre: %s\nTeléfono: %s" % (resultado)) ❸
cursor.close()
conexión.close()
```

El programa es muy parecido al anterior, una vez que necesitamos importar el módulo del SQLite; establecer una conexión y crear un cursor. El comando SQL que realiza una consulta es el comando **select**.

```
select * from agenda
```

El comando **select**, en su forma más simple, utiliza una lista de campos y una lista de tablas. En nuestro ejemplo, la lista de campos fue sustituida por ***** (asterisco). El asterisco representa todos los campos de la tabla siendo consultados, en este caso **nombre** y **teléfono**. La palabra **from** es utilizada para separar la lista de campos de la lista de tablas. En nuestro ejemplo, solo la tabla agenda. El comando **select** es ejecutado en la línea ❶.

Para acceder a los resultados del comando **select**, debemos utilizar el método **fetchone** de nuestro cursor ❷. Ese método devuelve una tupla con los resultados de nuestra consulta o **None**, en caso que la tabla esté vacía. Para simplificar nuestro ejemplo, el test de **None** fue retirado.

La tupla retornada tiene el mismo orden de los campos de nuestra consulta, en este caso **nombre** y **teléfono**. Así, **resultado[0]** es el primer campo, en el caso **nombre** y **resultado[1]** es el segundo, **teléfono**. En ❸, usamos una cadena de caracteres en Python y una máscara con dos **%s**, una para cada campo en la tupla **resultado**.

Ejecute el programa y verifique el resultado:

```
Nombre: Nilo
```

```
Teléfono: 7788-1432
```

Veamos ahora cómo incluir los otros teléfonos de nuestra agenda. El programa de la lista 11.3 presenta el método **executemany**. La principal diferencia entre **executemany** y **execute** es que **executemany** trabaja con varios valores. En nuestro ejemplo, utilizamos una lista de tuplas, **datos**. Cada elemento de la lista es una tupla con dos valores, exactamente como hicimos en el programa de la lista en 11.1.

► Lista 11.3 – Introduciendo múltiples registros

```
import sqlite3
datos = [("Juan", "98901-0109"),
```

```
("André", "98902-8900"),  
("María", "97891-3321")]
```

```
conexión = sqlite3.connect("agenda.db")  
cursor = conexión.cursor()  
cursor.executemany('''  
    insert into agenda (nombre, teléfono)  
    values(?, ?)  
    ''', datos)  
conexión.commit()  
cursor.close()  
conexión.close()
```

Con los datos introducidos por el programa, nuestra agenda debe tener ahora 4 registros. Veamos cómo imprimir el contenido de nuestra tabla, usando el mismo comando SQL, pero esta vez trabajando con varios resultados.

► Lista 11.4 – Consulta con múltiples resultados

```
import sqlite3  
conexión = sqlite3.connect("agenda.db")  
cursor = conexión.cursor()  
cursor.execute("select * from agenda")  
resultado=cursor.fetchall() ❶  
for registro in resultado:  
    print("Nombre: %s\nTeléfono: %s" % (registro)) ❷  
cursor.close()  
conexión.close()
```

Vea el nuevo programa de consulta en la lista 11.4. En ❶, utilizamos el método **fetchall** de nuestro cursor para retornar una lista con los resultados de nuestra consulta. En ❷, utilizamos la variable **registro** para exhibir los datos. Así como vimos el método **executemany**, que acepta una lista de tuplas como parámetro, **fetchall** retorna una lista de tuplas. Cada elemento de esa lista es una tupla conteniendo todos los campos devueltos por la consulta. Una vez que tenemos la lista **resultado**, utilizamos un simple **for** para trabajar con cada registro.

El método **fetchall** retorna **None** en caso que el resultado de la consulta sea vacío. Veremos eso en otros ejemplos. Para consultas pequeñas, conteniendo pocos registros como resultado, el método **fetchall** es muy interesante y fácil de utilizar. Para consultas mayores, donde más de 100 registros son retornados, otros métodos de obtener los resultados de la consulta pueden ser más interesantes. Esos métodos evitan la creación de una larga lista que puede ocupar una gran cantidad de memoria, y demorar mucho tiempo en ser ejecutada.

La lista 11.5 muestra el método **fetchone** ❶ utilizado dentro de una estructura de repetición **while**. Como no sabemos cuántos registros serán devueltos, utilizamos un **while True**, que es interrumpido cuando el método **fetchone** retorna **None**, lo cual significa que todos los resultados de

la consulta ya fueron obtenidos. Puede leer “*fetch*” como “obtener”; por lo tanto, **fetchone** sería “obtener un resultado”; y **fetchall**, “obtener todos los resultados”. La ventaja de **fetchone** en este caso es que imprimimos cada resultado de la consulta apenas obtenemos uno y mantenemos la impresión a medida que otros resultados van llegando. Ese tiempo de entrega es un concepto importante a tener en cuenta, una vez que los datos pasan del banco a nuestro programa. Esa transferencia es controlada por el banco de datos, responsable de ejecutar nuestra consulta y generar los resultados.

► Lista 11.5 – Consulta, registro por registro

```
import sqlite3
conexión = sqlite3.connect("agenda.db")
cursor = conexión.cursor()
cursor.execute("select * from agenda")
while True:
    resultado=cursor.fetchone() ❶
    if resultado == None:
        break
    print("Nombre: %s\nTeléfono: %s" % (resultado))
cursor.close()
conexión.close()
```

Antes de pasar a comandos SQL más avanzados, veamos la estructura **with** de Python que puede ayudarnos a no olvidarnos de llamar los métodos **close** de nuestros objetos. La lista 11.6 muestra el programa equivalente al de la lista 11.5, pero utilizando la cláusula **with**. Una de las ventajas de utilizar **with** es que creamos un bloque donde un objeto es considerado como válido. Si algo sucede dentro del bloque, como una excepción, la estructura **with** garantiza que el método **close** será llamado. En realidad, **with** llama al método **__exit__** en el final del bloque y funciona muy bien con archivos y conexiones de banco de datos. Lamentablemente, los cursores no tienen el método **__exit__**, obligándonos a llamar manualmente el método **close**, o a importar un módulo especial, **contextlib**, que ofrece la función **closing** ❶, la cual adapta un cursor con un método **__exit__**, que llama a **close**. Por ahora ese detalle puede quedar solo como una curiosidad, pero hablaremos más de **with** en el resto de este capítulo.

► Lista 11.6 – Uso del with para cerrar la conexión

```
import sqlite3
from contextlib import closing ❶
with sqlite3.connect("agenda.db") as conexión:
    with closing(conexión.cursor()) as cursor:
        cursor.execute("select * from agenda")
        while True:
            resultado=cursor.fetchone()
            if resultado == None:
```

```
break
```

```
print("Nombre: %s\nTeléfono: %s" % (resultado))
```

Ejercicio 11.1 Haga un programa que cree el banco de datos *precios.db* con la tabla **precios** para almacenar una lista de precios de venta de productos. La tabla debe contener el nombre del producto y su respectivo precio. El programa también debe introducir algunos datos para test.

Ejercicio 11.2 Haga un programa para listar todos los precios del banco *precios.db*.

11.4 Consultando registros

Hasta ahora no fuimos más allá de lo que podríamos haber hecho con simples archivos texto. La utilidad de un sistema gestor de base de datos comienza a aparecer cuando necesitamos buscar y alterar datos. Al trabajar con archivos, esas operaciones deben ser implementadas en nuestros programas, pero con el SQLite, podemos realizarlas usando comandos SQL. Primero, vamos a utilizar una variación del comando **select** para mostrar solo algunos registros, implementando una selección de registros basada en una investigación. Las investigaciones en SQL son hechas con la cláusula **where**. Veamos el comando SQL que selecciona todos los registros de la agenda, cuyo nombre sea igual a “Nilo”.

```
select * from agenda where nombre = "Nilo"
```

Vea que solo agregamos la cláusula **where** después del nombre de la tabla. El criterio de selección o de investigación debe ser escrito como una expresión, en el caso **nombre = "Nilo"**. La lista 11.7 muestra el programa con esa modificación.

► Lista 11.7 – Consulta con filtro de selección

```
import sqlite3
conexión = sqlite3.connect("agenda.db")
cursor = conexión.cursor()
cursor.execute("select * from agenda where nombre = 'Nilo'")
while True:
    resultado = cursor.fetchone()
    if resultado == None:
        break
print("Nombre: %s\nTeléfono: %s" % (resultado))
cursor.close()
conexión.close()
```

Al ejecutar el programa de la lista 11.7, debemos tener solo un resultado:

```
Nombre: Nilo
```

```
Teléfono: 7788-1432
```

Vea que escribimos 'Nilo' entre apóstrofes. Aquí, podemos usar un poco de lo que ya sabemos

sobre cadenas de caracteres en Python y escribir:

```
cursor.execute('select * from agenda where nombre = "Nilo"')
```

O sea, podríamos cambiar las comillas por apóstrofes o aun usar comillas triples:

```
cursor.execute("""select * from agenda where nombre = "Nilo" """)
```

En el caso de nuestro ejemplo, el nombre 'Nilo' es una constante y no hay problema en escribirlo directamente en nuestro comando **select**. Mientras tanto, en caso que el nombre a filtrar viniese de una variable, estaríamos tentados de escribir un programa, como el de la lista 11.8.

► Lista 11.8 – Consulta con filtro de selección surgido de variable

```
import sqlite3
nombre=input("Nombre a seleccionar: ")
conexión = sqlite3.connect("agenda.db")
cursor = conexión.cursor()
cursor.execute('select * from agenda where nombre = "%s"' % nombre)
while True:
    resultado=cursor.fetchone()
    if resultado == None:
        break
print("Nombre: %s\nTeléfono: %s" % (resultado))
cursor.close()
conexión.close()
```

Ejecute el programa de la lista 11.8 con varios valores: Nilo, Juan y María. Pruebe también con un nombre que no exista. La cláusula **where** funciona de forma parecida a un filtro. Imagine que el comando **select** crea una lista y que la expresión lógica definida en el **where** es evaluada para cada elemento. Cuando el resultado de esa evaluación es verdadero, la línea es copiada para otra lista, la lista de resultados, devuelta como resultado de nuestra consulta.

Vea que el programa funciona relativamente muy bien, excepto cuando no encontramos nada y el programa termina sin decir mucho. Vamos a corregir ese problema luego, pero ejecute el programa de la lista 11.8 una vez más y digite la siguiente secuencia como nombre:

```
X" or "1"="1
```

¿Sorprendido con el resultado? Ese es el motivo para no utilizar variables en nuestras consultas. Ese tipo de vulnerabilidad es un ejemplo de SQLInjection, un ataque bien conocido. Esto sucede porque el comando SQL resultante es:

```
select * from agenda where nombre = "X" or "1"="1"
```

Para evitar este tipo de ataque, siempre utilice parámetros con valores variables.

El **or** del lenguaje SQL funciona de forma semejante al **or** de Python. De esa forma, nuestra entrada de datos fue modificada por un valor digitado en el programa. Ese tipo de error es muy grave y puede permanecer mucho tiempo en nuestros programas sin ser percibido. Eso sucede porque la consulta es una cadena de caracteres como cualquier otra, y el valor pasado por el método **execute**

es la cadena de caracteres resultante. De esa forma, el valor digitado por el usuario puede introducir elementos que nosotros no deseamos. Los operadores de comparación **and** y **not** funcionan exactamente como en Python, y usted también puede usarlos en expresiones SQL.

Para no caer en ese tipo de trampa, utilizaremos siempre parámetros en nuestras consultas.

► Lista 11.9 – Consulta utilizando parámetros

```
import sqlite3

nombre = input("Nombre a seleccionar: ")
conexión = sqlite3.connect("agenda.db")
cursor = conexión.cursor()
cursor.execute('select * from agenda where nombre = ?', (nombre,))
x = 0
while True:
    resultado = cursor.fetchone()
    if resultado == None:
        if x == 0:
            print("Nada encontrado.")
            break
    print("Nombre: %s\nTeléfono: %s" % (resultado))
    x += 1
cursor.close()
conexión.close()
```

En la lista 11.9, utilizamos un parámetro, como hicimos antes para introducir nuestros registros. Un detalle importante es que escribimos **(nombre,)**, note la coma después de **nombre**. Ese detalle es importante, pues el segundo parámetro del método **execute** es una tupla, y en Python, las tuplas con un solo elemento son escritas con una coma después del primer valor. Vea también que utilizamos la variable **x** para contar cuántos resultados obtuvimos. Como el método **fetchone** retorna **None** cuando todos los registros fueron recibidos, verificamos si **x == 0**, para saber si algo ya había sido obtenido anteriormente o si debemos imprimir un mensaje diciendo que nada fue encontrado.

Ejercicio 11.3 Escriba un programa que realice consultas del banco de datos `precios.db`, creado en el ejercicio 11.1. El programa debe preguntar el nombre del producto y listar su precio.

Ejercicio 11.4 Modifique el programa del ejercicio 11.3 de modo de preguntar dos valores y listar todos los productos con precios entre esos dos valores.

11.5 Actualizando registros

Ya sabemos cómo crear tablas, introducir registros y hacer consultas simples. Vamos a empezar a

usar el comando **update** para alterar nuestros registros. Por ejemplo, vamos a alterar el registro con el teléfono de "Nilo" a "12345-6789":

```
update agenda set teléfono = "12345-6789" where nombre = 'Nilo'
```

La cláusula **where** funciona como en el comando **select**, o sea, evalúa una expresión lógica que, cuando es verdadera, incluye el registro en la lista de registros a modificar. La segunda parte del comando **update** es la cláusula **set**. Esa cláusula es usada para indicar qué hacer en los registros seleccionados por la expresión del **where**. En el ejemplo, **set teléfono = "12345-6789"** cambia el contenido del campo **teléfono** para "12345-6789". El comando entero podría ser leído como: actualice los registros de la tabla agenda, cambiando el teléfono a "12345-6789" en todos los registros donde el campo **nombre** es igual a "Nilo". Veamos el programa de la lista 11.10.

► Lista 11.10 – Actualizando el teléfono

```
import sqlite3
conexión = sqlite3.connect("agenda.db")
cursor = conexión.cursor()
cursor.execute("""update agenda
                set teléfono = '12345-6789'
                where nombre = 'Nilo'""")
conexión.commit()
conexión.close()
```

En este ejemplo utilizamos constantes, por lo tanto no necesitamos usar parámetros. Las mismas reglas que aprendimos para el comando **select** se aplican al comando **update**. Si los valores no son constantes, usted tiene que utilizar parámetros.

El comando **update** puede alterar más de un registro de una sola vez. Haga una copia del archivo **agenda.db** y pruebe modificar el programa de la lista 11.10, retirando la cláusula **where**:

```
update agenda set teléfono = "12345-6789"
```

usted verá que todos los registros fueron modificados:

```
Nombre: Nilo
Teléfono: 12345-6789
Nombre: Juan
Teléfono: 12345-6789
Nombre: André
Teléfono: 12345-6789
Nombre: María
Teléfono: 12345-6789
```

Sin la cláusula **where**, todos los registros serán seleccionados y alterados. Vamos a utilizar la propiedad **rowcount** de nuestro cursor para saber cuántos registros fueron alterados por nuestro **update**. Vea el programa de la lista 11.11 con esas alteraciones.

► Lista 11.11 – Ejemplo de update sin where y con rowcount


```

import sqlite3
conexión = sqlite3.connect("agenda.db")
cursor = conexión.cursor()
cursor.execute("""update agenda
                set teléfono = '12345-6789' """)
print("Registros alterados: ", cursor.rowcount)
conexión.commit()
conexión.close()

```

No se olvide que después de modificar el banco de datos, necesitamos llamar al método **commit**, como hicimos al introducir los registros. En caso que nos olvidemos, los cambios se perderán.

La propiedad **rowcount** es muy interesante para confirmar el resultado de comandos de actualización, como **update**. Esa propiedad no funciona con **select**, retornando siempre **-1**. Por eso, en la lista 11.9, contamos los registros retornados por nuestro **select** en vez de usar **rowcount**. En el caso de **update**, podríamos hacer una verificación de cuantos registros serían alterados antes de llamar a **commit**. Veamos el programa de la lista 11.12.

► Lista 11.12 – Update con rollback

```

import sqlite3
conexión = sqlite3.connect("agenda.db")
cursor = conexión.cursor()
cursor.execute("""update agenda
                set teléfono = '12345-6789' """)
print("Registros alterados: ", cursor.rowcount)
if cursor.rowcount == 1:
    conexión.commit()
    print("Alteraciones grabadas")
else:
    conexión.rollback()
    print("Alteraciones abortadas")
conexión.close()

```

En el programa de la lista 11.12, utilizamos el valor de **rowcount** para decidir si las alteraciones deberían ser registradas o ignoradas. Como ya sabemos, el método **commit** graba las alteraciones o cambios. El método **rollback** hace lo inverso, abortando las alteraciones y dejando el banco de datos como antes. Los métodos **commit** y **rollback** se ocupan del control de transacciones del banco de datos. Podemos entender una transacción como un conjunto de operaciones que debe ser ejecutado completamente. Eso significa operaciones que no tienen sentido, a menos que sean realizadas en un solo grupo. Si la ejecución del grupo falla, todas las alteraciones causadas durante a transacción corriente deben ser revertidas (**rollback**). En caso que todo ocurra como fue planeado, las operaciones serán almacenadas definitivamente en el banco de datos (**commit**). Veremos otros ejemplos más adelante.

Ejercicio 11.5 Escriba un programa que aumente el precio de todos los productos del banco *precios.db* en 10%.

Ejercicio 11.6 Escriba un programa que pregunte el nombre del producto y un nuevo precio. Usando el banco *precios.db*, actualice el precio de este producto en el banco de datos.

11.6 Borrando registros

Además de introducir, consultar y alterar registros, también podemos borrarlos. El comando **delete** borra registros basado en un criterio de selección, especificado en la cláusula **where** que ya conocemos. Haga otra copia del archivo *agenda.db*. Copie el antiguo banco de datos, con los registros previos a la ejecución del programa de la lista 11.11.

La sintaxis del comando **delete** es:

```
delete from agenda where nombre = 'María'
```

En otras palabras, borre de la tabla *agenda* todos los registros con nombre igual a "María". Veamos el programa de la lista 11.13.

► Lista 11.13 – Borrando registros

```
import sqlite3
conexión = sqlite3.connect("agenda.db")
cursor = conexión.cursor()
cursor.execute("""delete from agenda
                where nombre = 'María' """)
print("Registros borrados: ", cursor.rowcount)
if cursor.rowcount == 1:
    conexión.commit()
    print("Alteraciones grabadas")
else:
    conexión.rollback()
    print("Alteraciones abortadas")
conexión.close()
```

Utilizamos el método **rowcount** para tener certeza que estábamos borrando solo un registro. Así como con los comandos **insert** y **update**, necesita llamar **commit** para grabar las alteraciones o **rollback**, en caso contrario.

11.7 Simplificando el acceso sin cursores

La interfaz de banco de datos de Python nos permite ejecutar algunos comandos utilizando directamente el objeto de la conexión, sin crear explícitamente un cursor. Veamos la lista 11.14, que es una versión simplificada del programa de la lista 11.4.

► Lista 11.14 – Consulta de varios registros, acceso simplificado

```
import sqlite3
with sqlite3.connect("agenda.db") as conexión:
    for registro in conexión.execute("select * from agenda"): ❶
        print("Nombre: %s\nTeléfono: %s" % (registro))
```

En la lista 11.14, utilizamos la estructura `with` para facilitar el cierre de la conexión. En ❶, `conexión.execute` retorna un cursor que puede ser usado con `for`. Puede también utilizar el método `executemany` directamente con el objeto `conexión`. Esa opción simplificada funciona muy bien con SQLite, pero no forma parte de la interfaz estándar de banco de datos de Python, a DB-API 2.0. Al utilizar cursores, usted obedece a DB-API 2.0 que es implementada por otros bancos de datos, simplificando la migración de su código a otros bancos de datos, como el MySQL o MariaDB.

11.8 Accediendo a los campos como en un diccionario

Acceder a los campos por posición no siempre es tan fácil. En Python, usando SQLite, podemos acceder por el nombre, agregando una línea:

```
conexión.row_factory = sqlite3.Row
```

Veamos el programa completo en la lista 11.15.

► Lista 11.15 – Accediendo a los campos por el nombre

```
import sqlite3
conexión = sqlite3.connect("agenda.db")
conexión.row_factory = sqlite3.Row
cursor = conexión.cursor()
for registro in cursor.execute("select * from agenda"):
    print("Nombre: %s\nTeléfono: %s" % (registro["nombre"], registro["teléfono"]))
cursor.close()
conexión.close()
```

De esa forma, un registro puede ser accedido como si fuese un diccionario, donde el nombre del campo es usado como clave. Otra facilidad que esa línea tiene es que las claves son aceptadas independientemente de si se escribe el nombre de los campos en mayúsculas o minúsculas. Por ejemplo:

```
print("Nombre: %s\nTeléfono: %s" % (registro["NOMBRE"], registro["Teléfono"]))
```

11.9 Generando una clave primaria

Hasta ahora trabajamos solo con campos normales, con campos que contienen datos. A medida que crecen nuestras tablas, trabajar con los datos puede no ser la mejor solución, y necesitaremos agregar campos para mantener el banco de datos. Una de esas necesidades es identificar cada registro de manera única. Nosotros podemos utilizar datos que no se repiten, o que no deberían

repetirse, como el nombre de la persona, como una clave primaria. Podemos entender una clave primaria como la clave de un diccionario pero, en este caso, para tablas en nuestro banco de datos. Cualquier campo o un conjunto de campos puede servir de clave primaria. Una alternativa ofrecida por el SQLite es la generación automática de claves. En tal caso, el banco se encarga de crear números únicos para cada registro.

Vamos a implementar otro banco de datos, con la población de cada estado del Brasil. Veremos cómo dejar que el SQLite genere una clave primaria automáticamente:

```
create table estados(  
    id integer primary key autoincrement,  
    nombre text,  
    población integer)
```

Al crear la tabla `estados`, estamos especificando tres campos: `id`, `nombre` y `población`. Vea que `id` y `población` son del tipo `integer`, o sea, números enteros (`int`). `id` es el campo que elegimos para ser la clave primaria de esa tabla, y escribimos `primary key autoincrement` para que el SQLite genere esos números automáticamente. Entienda `id` como la abreviación de “identificador único” o “identidad”. *Primary key* significa clave primaria.

El programa de la lista 11.16 crea el banco de datos `brasil.db`, la tabla `estados` y también incluye el nombre y la población de todos los estados brasileños. Los datos fueron extraídos de la Wikipedia (https://pt.wikipedia.org/wiki/Lista_de_unidades_federativas_do_Brasil_por_popula%C3%A7%C3%A3o).

► Lista 11.16 – Creación del banco de datos con la población de los estados brasileños

```
import sqlite3  
  
datos = [("São Paulo",43663672), ("Minas Gerais",20593366), ("Rio de Janeiro",  
16369178), ("Bahia",15044127), ("Rio Grande do Sul",11164050), ("Paraná",10997462),  
("Pernambuco",9208511), ("Ceará",8778575), ("Pará",7969655), ("Maranhão",6794298),  
("Santa Catarina",6634250), ("Goiás",6434052), ("Paraíba", 3914418), ("Espírito  
Santo",3838363), ("Amazonas",3807923), ("Rio Grande do Norte", 3373960),  
("Alagoas", 3300938), ("Piauí",3184165), ("Mato Grosso",3182114), ("Distrito  
Federal",2789761), ("Mato Grosso do Sul",2587267), ("Sergipe",2195662),  
("Rondônia",1728214), ("Tocantins",1478163), ("Acre",776463), ("Amapá",734995),  
("Roraima",488072)]  
  
conexión = sqlite3.connect("brasil.db")  
conexión.row_factory = sqlite3.Row  
cursor = conexión.cursor()  
cursor.execute("""create table estados(  
    id integer primary key autoincrement,  
    nombre text,  
    población integer  
)""")  
  
cursor.executemany("insert into estados(nombre, población) values(?,?)", datos)  
conexión.commit()
```

```
cursor.close()
```

```
conexión.close()
```

El valor del campo `id` será generado automáticamente. Una vez que tenemos la población de los estados, vamos a hacer una consulta para listar los estados en orden alfabético. Vea el programa completo en la lista 11.17. Al ejecutar ese programa, observe los valores generados en el campo `id`, valores numéricos de 1 a 27 para este caso.

► Lista 11.17 – Consulta de los estados brasileños, ordenados por nombre

```
import sqlite3
conexión = sqlite3.connect("brasil.db")
conexión.row_factory = sqlite3.Row
print("%3s %-20s %12s" % ("Id", "Estado", "Población"))
print("="*37)
for estado in conexión.execute("select * from estados order by nombre"):
    print("%3d %-20s %12d" %
          (estado["id"],
           estado["nombre"],
           estado["población"]))
conexión.close()
```

La gran diferencia es que estamos utilizando la cláusula `order by` para ordenar los resultados de nuestra consulta; en este caso, por el campo `nombre`.

```
select * from estados order by nombre
```

Modifique el programa para que los estados se impriman de acuerdo a su población, usando la consulta:

```
select * from estados order by población
```

Ejecute el programa nuevamente y vea que los estados ahora fueron impresos de acuerdo a su población, pero de menor a mayor. Aunque ese sea el orden normal, cuando trabajamos con listas de estados por población, esperamos verlos ordenados empezando por estado más populoso al menos populoso, o sea, en el orden inverso (decreciente) de los valores. Veamos ese resultado al adicionar `desc` después del nombre del campo:

```
select * from estados order by población desc
```

11.10 Alterando la tabla

Vamos a agregar algunos campos a nuestra tabla de estados. Un campo para la región del Brasil y otro para la sigla del estado. En SQL, el comando utilizado para alterar los campos de una tabla es el `alter table`.

```
alter table estados add sigla text
```

```
alter table estados add región text
```

El comando `alter table` del SQLite es limitado si se compara con otros bancos de datos. En otros bancos se pueden alterar varios campos con un solo `alter table`, pero en el SQLite, estamos obligados a alterar un campo por vez. Las limitaciones del `alter table` del SQLite no terminan ahí. Por eso, planeé sus tablas con cuidado y, en caso que precise realizar grandes cambios, es preferible crear otra tabla con las alteraciones y copiar los datos de la tabla antigua. Ejecute el programa de la lista 11.18 para modificar la tabla `estados` y adicionar los campos `sigla` y `región`.

► Lista 11.18 – Alterando la tabla

```
import sqlite3
with sqlite3.connect("brasil.db") as conexión:
    conexión.execute("""alter table estados
                        add sigla text""")
    conexión.execute("""alter table estados
                        add región text""")
```

Ahora que la tabla tiene los nuevos campos, vamos a alterar nuestros registros y llenar la `región` y `sigla` de cada estado. Ejecute el programa de la lista 11.19.

► Lista 11.19 – Llenando la sigla y la región de cada estado

```
import sqlite3
datos = [
    ["SP", "SE", "São Paulo"], ["MG", "SE", "Minas Gerais"], ["RJ", "SE", "Rio de Janeiro"],
    ["BA", "NE", "Bahia"], ["RS", "S", "Rio Grande do Sul"], ["PR", "S", "Paraná"],
    ["PE", "NE", "Pernambuco"], ["CE", "NE", "Ceará"], ["PA", "N", "Pará"],
    ["MA", "NE", "Maranhão"], ["SC", "S", "Santa Catarina"], ["GO", "CO", "Goiás"],
    ["PB", "NE", "Paraíba"], ["ES", "SE", "Espírito Santo"], ["AM", "N", "Amazonas"],
    ["RN", "NE", "Rio Grande do Norte"], ["AL", "NE", "Alagoas"], ["PI", "NE", "Piauí"],
    ["MT", "CO", "Mato Grosso"], ["DF", "CO", "Distrito Federal"], ["MS", "CO", "Mato Grosso do Sul"],
    ["SE", "NE", "Sergipe"], ["RO", "N", "Rondônia"], ["TO", "N", "Tocantins"],
    ["AC", "N", "Acre"], ["AP", "N", "Amapá"], ["RR", "N", "Roraima"]
]
with sqlite3.connect("brasil.db") as conexión:
    conexión.executemany("""update estados
                            set sigla = ?,
                                región = ?
                            where nombre = ?""", datos)
```

Ahora nuestro banco de datos tiene una tabla `estados` con la población, `sigla` y `región` de cada estado. Esos nuevos campos permitirán utilizar funciones de agregación del lenguaje SQL: `count`, `min`, `max`, `avg` y `sum`.

11.11 Agrupando datos

Un banco de datos puede realizar operaciones de agrupamiento de datos fácilmente. Podemos, por ejemplo, solicitar el valor mínimo de un grupo de registros, así como también el máximo o la media

de esos valores. Mientras tanto, tenemos que modificar nuestros comandos SQL para indicar una cláusula de agrupamiento, o sea, debemos indicar cómo el banco de datos debe agrupar nuestros registros.

Veamos cómo realizar un grupo simple y exhibir cuantos registros hacen parte de ese grupo, usando la función **count**. La cláusula SQL que indica agrupamiento es **group by**, seguida del nombre de los campos que componen el grupo. Imagine que el banco va a concatenar cada uno de esos campos, creando un valor para cada registro. Vamos a llamar a ese valor “clave de grupo”. Todos los registros con la misma clave de grupo forman parte del mismo grupo y serán representados por un solo registro en la consulta de selección. Esa consulta en grupo solo puede contener los campos utilizados para componer la clave del grupo y funciones de agrupamiento de datos, como **min** (mínimo), **max** (máximo), **avg** (media), **sum** (suma) y **count** (recuento).

Un ejemplo concreto como nuestro banco de datos es agrupar los estados por región. La consulta sería algo como:

```
select región, count(*) from estados group by región
```

Ese comando utiliza la cláusula **group by** región para especificar la clave de grupo. De esa forma, todos los registros que pertenecen a la misma región son agrupados. Observe que los campos después del **select** incluyen región y **count(*)**. El campo región pudo ser incluido pues es parte de la clave de grupo especificada en la **group by**. La función **count(*)** retorna cuántos registros forman parte del grupo. Veamos el resultado del programa de la lista 11.20.

► Lista 11.20 – Agrupando y contando estados por región

```
import sqlite3
print("Región Número de Estados")
print("=====  
with sqlite3.connect("brasil.db") as conexión:
    for región in conexión.execute("""
        select región, count(*)
        from estados
        group by región"""):
        print("{0:6} {1:17}".format(*región))
```

Resultado de la ejecución del programa de la lista 11.20:

Región Número de Estados

=====

CO 4

N 7

NE 9

S 3

SE 4

Vamos a agregar las funciones `min`, `max`, `sum`, `avg` en el campo población. Vea el programa de la lista 11.21.

► **Lista 11.21 – Usando las funciones de agregación**

```
import sqlite3
print("Región Estados Población Mínima Máxima Media Total (suma)")
print("===== ")
with sqlite3.connect("brasil.db") as conexión:
    for región in conexión.execute("""
        select región, count(*), min(población),
            max(población), avg(población), sum(población)
        from estados
        group by región"""):
        print("{0:6} {1:7} {2:18,} {3:10,} {4:10,.0f} {5:13,}".format(*región))
    print("\nBrasil: {0:6} {1:18,} {2:10,} {3:10,.0f} {4:13,}".format(
        *conexión.execute("""
            select count(*), min(población), max(población),
                avg(población), sum(población) from estados""").fetchone()))
```

Resultado del programa de la lista 11.21:

Región	Estados	Población	Mínima	Máxima	Media	Total (suma)
=====	=====	=====	=====	=====	=====	=====
CO	4	2,587,267	6,434,052	3,748,298	14,993,194	
N	7	488,072	7,969,655	2,426,212	16,983,485	
NE	9	2,195,662	15,044,127	6,199,406	55,794,654	
S	3	6,634,250	11,164,050	9,598,587	28,795,762	
SE	4	3,838,363	43,663,672	21,116,145	84,464,579	
Brasil:	27	488,072	43,663,672	7,445,618	201,031,674	

Con el programa de la lista 11.21, conseguimos calcular la población mínima, máxima, media y total de cada región y también de todo Brasil. Vea que en la segunda consulta, la que calcula los datos para Brasil, no utilizamos la cláusula `group by`, haciendo que todos los registros formen parte del grupo.

Al utilizar las funciones de agregación y la cláusula `group by`, podemos continuar usando todo lo que ya aprendimos en SQL, como las cláusulas `where` y `order by`. Veamos el mismo programa de la lista 11.21, pero con las líneas ordenadas por la población total de cada región, en orden decreciente.

► **Lista 11.22 – Funciones de agregación con order by**


```

import sqlite3

print("Región Estados Población Mínima Máxima Media Total (suma)")
print("===== ")

with sqlite3.connect("brasil.db") as conexión:
    for región in conexión.execute("""
        select región, count(*), min(población),
            max(población), avg(población), sum(población)
        from estados
        group by región
        order by sum(población) desc"""):
        print("{0:6} {1:7} {2:18,} {3:10,} {4:10,.0f} {5:13,}".format(*región))
    print("\nBrasil: {0:6} {1:18,} {2:10,} {3:10,.0f} {4:13,}".format(
        *conexión.execute("""
            select count(*), min(población), max(población),
                avg(población), sum(población) from estados""").fetchone()))

```

En el programa de la lista 11.22, solo agregamos la línea `order by sum(población) desc` en el final de nuestra consulta. Vea que repetimos la función de agregación `sum(población)` para indicar que el ordenamiento será hecho por la suma de la población. Puede utilizar la cláusula `as` del SQL para dar nombres a las columnas de una consulta. Vea la consulta modificada para usar `as` y crear una columna `tpop` para la suma de la población:

```

select región, count(*), min(población), max(población),
    avg(población), sum(población) as tpop
from estados group by región
order by tpop desc

```

Observe que escribimos `sum(población) as tpop`, dando el nombre `tpop` a la suma. Después utilizamos el nombre `tpop` en la cláusula del `order by`. Ese tipo de construcción evita la repetición de la función en la consulta y facilita la lectura.

Resultado de la ejecución del programa de la lista 11.22:

Región	Estados	Población	Mínima	Máxima	Media	Total (suma)
=====	=====		=====	=====	=====	=====
SE	4	3,838,363	43,663,672	21,116,145	84,464,579	
NE	9	2,195,662	15,044,127	6,199,406	55,794,654	
S	3	6,634,250	11,164,050	9,598,587	28,795,762	
N	7	488,072	7,969,655	2,426,212	16,983,485	
CO	4	2,587,267	6,434,052	3,748,298	14,993,194	

Brasil: 27 488,072 43,663,672 7,445,618 201,031,674

También podemos filtrar los resultados después del agrupamiento, usando la cláusula **having**. Para entender la diferencia entre **where** y **having**, imagine que **where** es ejecutada antes del agrupamiento, seleccionando los registros que formarán parte del resultado, antes de realizar el agrupamiento. La cláusula **having**, evalúa el resultado del agrupamiento y decide cuáles serán parte del resultado final. Por ejemplo, podemos elegir solo las regiones con más de 5 estados. Como la cantidad de estados por región solo es conocida después del agrupamiento (**group by**), esa condición debe aparecer en una cláusula **having**.

```
select región, count(*), min(población),
       max(población), avg(población), sum(población) as tpop
from estados group by región
having count(*)>5
order by tpop desc
```

Vea el programa completo en la lista 11.23.

► Lista 11.23 – Utilizando having para listar solo las regiones con más de 5 estados

```
import sqlite3

print("Región Estados Población  Mínima      Máxima      Media      Total (suma)")
print("===== =====  =====  =====  =====  =====")

with sqlite3.connect("brasil.db") as conexión:
    for región in conexión.execute("""
        select región, count(*), min(población),
               max(población), avg(población), sum(población) as tpop
        from estados
        group by región
        having count(*)>5
        order by tpop desc"""):
        print("{0:6} {1:7} {2:18,} {3:10,} {4:10,.0f} {5:13,}".format(*región))
```

Resultando en:

Región	Estados	Población	Mínima	Máxima	Media	Total (suma)
=====	=====		=====	=====	=====	=====
NE	9		2,195,662	15,044,127	6,199,406	55,794,654
N	7		488,072	7,969,655	2,426,212	16,983,485

Una vez que solo las regiones Norte (N) y Nordeste (NE) tienen más de 5 estados.

11.12 Trabajando con fechas

Aunque el SQLite trabaje con fechas, el tipo `DATE` no es soportado directamente, generando una cierta confusión entre fechas y cadenas de caracteres. Vamos a crear una tabla con un campo del tipo fecha

► Lista 11.24 – Creando una tabla de feriados nacionales

```
import sqlite3

feriados = [
    ["2014-01-01", "Confraternización Universal"],
    ["2014-04-21", "Tiradentes"],
    ["2014-05-01", "Día del trabajador"],
    ["2014-09-07", "Independencia"],
    ["2014-10-12", "Patrona del Brasil"],
    ["2014-11-02", "Día de los muertos"],
    ["2014-11-15", "Proclamación de la República"],
    ["2014-12-25", "Navidad"]
]

with sqlite3.connect("brasil.db") as conexión:
    conexión.execute("create table feriados(id integer primary key autoincrement,
        fecha date, descripción text)")
    conexión.executemany("insert into feriados(fecha,descripción) values (?,?)",
feriados)
```

En el programa de la lista 11.24, creamos la tabla `feriados` e introducimos algunas fechas. Observe que escribimos las fechas en el formato ISO 8601 (http://es.wikipedia.org/wiki/ISO_8601): AÑO-MES-DÍA. En ese formato, la comparación de Navidad (25/12/2014) es escrita como 2014-12-25. Escribir las fechas en ese formato es una característica del SQLite. Siempre escriba sus fechas en el formato ISO al trabajar con ese administrador de banco de datos. Observe que utilizamos el tipo `date` (fecha) en la columna comparación. Modifique el año de 2014 para el año corriente, en caso necesario.

Veamos cómo acceder a esos valores en el programa de la lista 11.25.

► Lista 11.25 – Accediendo a un campo del tipo fecha

```
import sqlite3

with sqlite3.connect("brasil.db") as conexión:
    for feriado in conexión.execute("select * from feriados"):
        print(feriado)
```

Que resulta en:

```
(1, '2014-01-01', 'Confraternización Universal')
(2, '2014-04-21', 'Tiradentes')
(3, '2014-05-01', 'Día del trabajador')
(4, '2014-09-07', 'Independencia')
(5, '2014-10-12', 'Patrona del Brasil')
(6, '2014-11-02', 'Día de los muertos')
(7, '2014-11-15', 'Proclamación de la República')
(8, '2014-12-25', 'Navidad')
```

En el programa de la lista 11.25, accedemos al campo `fecha` como hemos hecho hasta ahora, sin ningún procedimiento especial. Vea que al imprimir la tupla `feriado` con el resultado de nuestra selección, el campo `fecha` se imprimió como una cadena de caracteres cualquiera. Nada impide que utilicemos cadenas de caracteres para representar fechas, como hicimos al crear el banco de datos, pero los campos `fecha` son más interesantes, pues podemos consultar fácilmente el día de la semana y también realizar operaciones con fechas.

Vamos a modificar nuestra conexión con el SQLite de modo de solicitar el procesamiento de los tipos de campo en nuestras consultas. Al solicitar la conexión, debemos pasar `detect_types=sqlite3.PARSE_DECLTYPES` como parámetro. Veamos la lista 11.26 con esa modificación.

► Lista 11.26 – Solicitando el tratamiento del tipo de los campos

```
import sqlite3
with sqlite3.connect("brasil.db",detect_types=sqlite3.PARSE_DECLTYPES) as conexión:
    for feriado in conexión.execute("select * from feriados"):
        print(feriado)
```

El resultado del programa de la lista 11.26 es muy diferente:

```
(1, datetime.date(2014, 1, 1), 'Confraternización Universal')
(2, datetime.date(2014, 4, 21), 'Tiradentes')
(3, datetime.date(2014, 5, 1), 'Día del trabajador')
(4, datetime.date(2014, 9, 7), 'Independencia')
(5, datetime.date(2014, 10, 12), 'Patrona del Brasil')
(6, datetime.date(2014, 11, 2), 'Día de los muertos')
(7, datetime.date(2014, 11, 15), 'Proclamación de la República')
(8, datetime.date(2014, 12, 25), 'Navidad')
```

Vea que los valores del campo `fecha` ahora son objetos de la clase `datetime.date`. Eso evita tener

que hacer la conversión manualmente de una cadena de caracteres a `datetime.date`. En el programa de la lista 11.27, utilizamos el método `strftime` del objeto de la clase `datetime.date` para exhibir solo el día y el mes de la comparación, sin el año.

► Lista 11.27 – Trabajando con fechas

```
import sqlite3
with sqlite3.connect("brasil.db",detect_types=sqlite3.PARSE_DECLTYPES) as conexión:
    conexión.row_factory = sqlite3.Row
    for feriado in conexión.execute("select * from feriados"):
        print("{0} {1}".format(feriado["fecha"].strftime("%d/%m"), feriado["descripción"]))
```

En el programa de la lista 11.27, volvemos a utilizar `row_factory` para acceder a los campos por nombre, como en un diccionario. El método `strftime` fue utilizado con la máscara `"%d/%m"` para exhibir solo el día y el mes. Puede verificar los formatos aceptados por `strftime` en la tabla 9.3.

Veamos el resultado del programa de la lista 11.27:

```
01/01 Confraternización Universal
21/04 Tiradentes
01/05 Día del trabajador
07/09 Independencia
12/10 Patrona del Brasil
02/11 Día de los muertos
15/11 Proclamación de la República
25/12 Navidad
```

Veamos un poco lo que podemos hacer con los objetos del módulo `datetime`.

► Lista 11.28 – Feriados en los próximos 60 días

```
import sqlite3
import datetime
hoy = datetime.date.today()
hoy60días = hoy + datetime.timedelta(days=60)
with sqlite3.connect("brasil.db",detect_types=sqlite3.PARSE_DECLTYPES) as conexión:
    conexión.row_factory = sqlite3.Row
    for feriado in conexión.execute("select * from feriados where fecha >= ? and
    fecha <= ?", (hoy, hoy60días)):
        print("{0} {1}".format(feriado["fecha"].strftime("%d/%m"),
    feriado["descripción"]))
```

El programa de la lista 11.28 utiliza objetos del módulo `datetime`. En `hoy`, guardamos la comparación actual (`datetime.date.today()`). En `hoy60días`, utilizamos un objeto del tipo `datetime.timedelta` para incrementar en 60 días la comparación actual. Con esos dos objetos `date`, podemos utilizar la cláusula `where` del SQLite para seleccionar los feriados entre `hoy` y

hoy60días. Consulte la documentación de Python para saber más sobre el módulo `datetime`. Los objetos de las clases `timedelta` y `datetime.datetime` son bastante útiles, en caso que usted precise realizar operaciones con fechas y guardar la hora con la comparación (`datetime`).

11.13 Claves y relaciones

Ahora que ya sabemos lo básico de cómo manipular registros en nuestro banco de datos, veremos conceptos más avanzados que nos permitirán trabajar con varias tablas. Para seleccionar nuestros registros, vimos que necesitamos construir expresiones lógicas que identifiquen o que permitan la selección de esos registros. En el caso de nuestra agenda, el campo `nombre` fue usado en nuestras expresiones, pero utilizar datos como criterio de selección no es una buena idea a largo plazo. Los datos pueden cambiar y repetirse entre varias tablas. Por ejemplo, imagine una situación donde nuestra agenda tenga varios teléfonos por persona, como en la tabla 11.2:

Tabla 11.2 – Agenda con una tabla

nombre	número	tipo
Nilo	12345-6789	Casa
Nilo	98745-4321	Celular

En ese ejemplo, podríamos simplemente adicionar dos registros con el mismo nombre, pero estaríamos complicando nuestro trabajo más tarde, pues si quisiésemos cambiar el teléfono de uno de los registros, no podríamos utilizar `nombre = "Nilo"` como criterio de selección, nos veríamos obligados a utilizar una condición compuesta por `nombre` y `teléfono`.

Ese problema podría resolverse adicionando una clave que identificase cada persona de forma única. Esa clave puede ser un simple número, basta que sea único, y vamos a llamarla identificador o simplemente `id`. La tabla 11.3 muestra nuestros datos con ese nuevo campo.

Tabla 11.3 – Agenda con una tabla y una clave

id	nombre	número	tipo
1	Nilo	12345-6789	Casa
1	Nilo	98745-4321	Celular

Aunque nuestros datos estén en mejor forma, aún estamos repitiendo el campo `nombre` varias veces. Y el campo `id` no identifica únicamente cada registro. Ese tipo de problema es llamado redundancia de datos. En una base de datos, cuanto menos redundancias tengamos en nuestros datos, más sencillo será su mantenimiento. Por ejemplo, imagine que queremos cambiar el nombre Nilo para agregar también Menezes. Tendríamos que actualizar los dos registros, pues guardamos la misma información en dos lugares (registros) diferentes.

Una mejor forma de representar esos datos es dividiendo nuestros datos en varias tablas. Por ejemplo, una tabla para nombre y otra para teléfono. Veamos las tablas 11.4 y 11.5 con esa nueva división.

Tabla 11.4 – Tabla nombres

id	Nombre

Tabla 11.5 – Tabla teléfonos

id_nombre	número	tipo
1	12345-6789	Casa
1	98745-4321	Celular

Vea que el campo **id** en la tabla 11.4 puede ser usado como clave primaria. La clave primaria (**id**) de la tabla nombres fue copiada en la tabla 11.5, en el campo **id_nombre**.

De esa forma almacenamos el **nombre** en un solo lugar. Aún tenemos otros problemas, pues ahora nuestros teléfonos no tienen un identificador único. Vea cómo quedaría nuestra tabla, si agregamos una clave a teléfonos, tabla 11.6:

Tabla 11.6 – Tabla teléfonos

id	id_nombre	número	tipo
1	1	12345-6789	Casa
2	1	98745-4321	Celular

Así, una clave primaria es un campo de un registro que lo identifica de forma única en la tabla. Resolvemos nuestro problema de redundancia, ¿Pero como acceder a esos datos en tablas diferentes con comandos SQL? Bien, vamos a utilizar el comando **select**, pero con varias tablas y vamos a especificar una forma de relacionarlas.

```
select * from nombres, teléfonos where nombres.id = teléfonos.id_nombres
```

Ese comando difiere de nuestros otros ejemplos por utilizar más de una tabla, después de la cláusula **from**. Una vez que utilizamos varias tablas, estamos obligados a especificar cómo se relacionan esas tablas; en caso contrario obtendremos lo que es llamado producto cartesiano, donde nuestro resultado contendrá la combinación de cada registro de la primera tabla, con cada registro de la segunda. Es ese el relacionamiento especificado en **nombres.id = teléfonos.id_nombres**. En otras palabras, especificamos un criterio que vincula las dos tablas cuando el campo **id** de la tabla nombres (**nombres.id**) es igual al campo **id_nombres** de la tabla teléfonos (**teléfonos.id_nombres**).

Pero el tipo del teléfono aún se repite, lo que puede llevar a resultados indeseables en nuestra agenda. Vamos a crear otra tabla para guardar los tipos de teléfono de modo de no repetirlos. Vea el resultado en las tablas 11.7 y 11.8.

Tabla 11.7 – Tabla teléfonos con el campo id_tipo

id	id_nombre	número	id_tipo
1	1	12345-6789	1
2	1	98745-4321	2

Tabla 11.8 – Tabla tipos

id	descripción
1	Casa
2	Celular

Veamos los comandos SQL para crear esas tablas:

```
create table tipos(id integer primary key autoincrement,  
                  descripción text);  
create table nombres(id integer primary key autoincrement,  
                    nombre text);  
create table teléfonos(id integer primary key autoincrement,  
                      id_nombre integer,  
                      número text,  
                      id_tipo integer);
```

Con el uso de `primary key autoincrement`, como ya vimos anteriormente, el SQLite se encargará de generar los números que utilizaremos en nuestras claves primarias. Ahora podemos revisar la agenda del capítulo 10 y convertirla para utilizar un banco de datos en vez de un simple archivo de texto.

11.14 Convirtiendo la agenda para utilizar un banco de datos

Convertir la agenda para utilizar un banco de datos nos llevará a enfrentar un problema de mapeo entre objetos y los bancos de datos de comparación, como el SQLite. Uno de los mayores problemas en ese tipo de mapeo es mantener sincronizados los datos entre nuestro programa y el banco de datos. Existen bibliotecas enteras escritas únicamente para resolver ese tipo de problemas, usando lo que se llama Mapeo Objeto Relacional (*Object-Relational Mapping*, ORM).

En nuestra agenda tenemos una lista de registros, cada uno con un nombre y una lista de teléfonos. Cada teléfono tiene un tipo pre-registrado.

Primero vamos a crear una subclase de `ListaÚnica` para controlar los registros borrados de nuestras listas. Después, crearemos métodos en otra clase, llamada `DBAgenda`, responsable de mantener el banco de datos y ejecutar las operaciones de la agenda. Un cambio en esa nueva versión de la agenda es que cargamos el registro solo cuando necesitamos cargarlo. Al volver al menú principal, todos los cambios ya estarán guardados en el banco de datos, volviendo inútiles las opciones Lee y Graba.

► Lista 11.29 – Nuevas clases – lista parcial

```
class DBListaÚnica(ListaÚnica):  
    def __init__(self, elem_class):  
        super().__init__(elem_class)  
        self.borrados = []  
    def remove(self, elem):  
        if elem.id is not None:  
            self.borrados.append(elem.id)  
            super().remove(elem)  
    def limpia(self):  
        self.borrados = []
```



```

class DBNombre(Nombre):
    def __init__(self, nombre, id_=None):
        super().__init__(nombre)
        self.id = id_

class DBTipoTeléfono(TipoTeléfono):
    def __init__(self, id_, tipo):
        super().__init__(tipo)
        self.id = id_

class DBTeléfono(Teléfono):
    def __init__(self, número, tipo=None, id_=None, id_nombre=None):
        super().__init__(número, tipo)
        self.id = id_
        self.id_nombre = id_nombre

class DBTeléfonos(DBListaÚnica):
    def __init__(self):
        super().__init__(DBTeléfono)

class DBTiposTeléfono(ListaÚnica):
    def __init__(self):
        super().__init__(DBTipoTeléfono)

class DBDatoAgenda:
    def __init__(self, nombre):
        self.nombre = nombre
        self.teléfonos = DBTeléfonos()

    @property
    def nombre(self):
        return self.__nombre

    @nombre.setter
    def nombre(self, valor):
        if type(valor) != DBNombre:
            raise TypeError("nombre debe ser una instancia de la clase DBNombre")
        self.__nombre = valor

    def investigaciónTeléfono(self, teléfono):
        posición = self.teléfonos.investigación(DBTeléfono(teléfono))
        if posición == -1:
            return None
        else:
            return self.teléfonos[posición]

```

La clase `DBListaÚnica` hereda de nuestra clase `ListaÚnica` y su principal función es mantener una

lista de `id` borrados. Eso nos permitirá borrar los elementos de nuestras listas y, en una fase siguiente, borrarlos del banco de datos. La clase `DBListaÚnica` solo puede trabajar con clases que tengan un atributo `id`.

Las clases `DBNombre` y `DBTeléfono` derivan de `Nombre` y `Teléfono` respectivamente. La principal diferencia es que ahora incluyen el atributo `id`. Vea que ese atributo es un parámetro opcional, pues al crear nuestros objetos, ellos no tendrán aún sus claves primarias. Además de eso, tendremos en cuenta el hecho que probablemente acaban de ser creados objetos sin `id` y necesitan ser introducidos en el banco de datos. Eso quedará más claro en el programa completo.

Ya en la clase `DBDatoAgenda` modificamos el tipo de la lista de teléfonos de `Teléfonos` para `DBTeléfonos`. La clase `DBTeléfonos` es una derivación de `DBListaÚnica` que acepta solo elementos del tipo `DBTeléfono`. Hicimos lo mismo entre `DBTipoTeléfono` y `DBTiposTeléfonos`.

Hasta ahora, hicimos solo el cambio de los tipos, como preparación para trabajar con el banco de datos. El principal cambio fue el nuevo atributo `id` que agregamos en todas nuestras clases. Es el valor de ese campo el que utilizaremos en nuestras consultas, alteraciones y remociones.

► Lista 11.30 – Lista parcial – Clase `DBAgenda`

```
BANCO = ""

create table tipos(id integer primary key autoincrement,
    descripción text);
create table nombres(id integer primary key autoincrement, nombre text);
create table teléfonos(id integer primary key autoincrement,
    id_nombre integer,
    número text,
    id_tipo integer);
insert into tipos(descripción) values ("Celular");
insert into tipos(descripción) values ("Fijo");
insert into tipos(descripción) values ("Fax");
insert into tipos(descripción) values ("Casa");
insert into tipos(descripción) values ("Trabajo");
""

class DBAgenda:
    def __init__(self, banco):
        self.tiposTeléfono = DBTiposTeléfono()
        self.banco = banco
        nuevo = not os.path.isfile(banco)
        self.conexión = sqlite3.connect(banco)
        self.conexión.row_factory = sqlite3.Row
        if nuevo:
            self.crea_banco()
        self.cargaTipos()
```

```

def cargaTipos(self):
    for tipo in self.conexión.execute("select * from tipos"):
        id_ = tipo["id"]
        descripción = tipo["descripción"]
        self.tiposTeléfono.adiciona(DBTipoTeléfono(id_, descripción))

def crea_banco(self):
    self.conexión.executescript(BANCO)

def investigaciónNombre(self, nombre):
    if not isinstance(nombre, DBNombre):
        raise TypeError("nombre debe ser del tipo DBNombre")
    encontrado = self.conexión.execute("""select count(*)
        from nombres where nombre = ?""",
        (nombre.nombre,)).fetchone()
    if(encontrado[0]>0):
        return self.carga_por_nombre(nombre)
    else:
        return None

def carga_por_id(self, id):
    consulta = self.conexión.execute(
        "select * from nombres where id = ?", (id,))
    return carga(consulta.fetchone())

def carga_por_nombre(self, nombre):
    consulta = self.conexión.execute(
        "select * from nombres where nombre = ?", (nombre.nombre,))
    return self.carga(consulta.fetchone())

def carga(self, consulta):
    if consulta is None:
        return None
    nuevo = DBDatoAgenda(DBNombre(consulta["nombre"], consulta["id"]))
    for teléfono in self.conexión.execute(
        "select * from teléfonos where id_nombre = ?",
        (nuevo.nombre.id,)):
        ntel = DBTeléfono(teléfono["número"], None,
            teléfono["id"], teléfono["id_nombre"])
    for tipo in self.tiposTeléfono:
        if tipo.id == teléfono["id_tipo"]:
            ntel.tipo = tipo
            break
    nuevo.teléfonos.adiciona(ntel)

```

```

    return nuevo
def lista(self):
    consulta = self.conexión.execute(
        "select * from nombres order by nombre")
    for registro in consulta:
        yield self.carga(registro)
def nuevo(self, registro):
    try:
        cur = self.conexión.cursor()
        cur.execute("insert into nombres(nombre) values (?)",
            (str(registro.nombre),))
        registro.nombre.id = cur.lastrowid
        for teléfono in registro.teléfonos:
            cur.execute("""insert into teléfonos(número,
                id_tipo, id_nombre) values (?, ?, ?)""",
                (teléfono.número, teléfono.tipo.id,
                 registro.nombre.id))
            teléfono.id = cur.lastrowid
        self.conexión.commit()
    except:
        self.conexión.rollback()
        raise
    finally:
        cur.close()
def actualiza(self, registro):
    try:
        cur = self.conexión.cursor()
        cur.execute("update nombres set nombre=? where id = ?",
            (str(registro.nombre), registro.nombre.id))
        for teléfono in registro.teléfonos:
            if teléfono.id is None:
                cur.execute("""insert into teléfonos(número,
                    id_tipo, id_nombre)
                    values (?, ?, ?)""",
                    (teléfono.número, teléfono.tipo.id,
                     registro.nombre.id))
                teléfono.id = cur.lastrowid
            else:

```

```

        cur.execute("""update teléfonos set número=?,
                        id_tipo=?, id_nombre=?
                        where id = ?""",
                    (teléfono.número, teléfono.tipo.id,
                     registro.nombre.id, teléfono.id))
    for borrado in registro.teléfonos.borrados:
        cur.execute("delete from teléfonos where id = ?", (borrado,))
    self.conexión.commit()
    registro.teléfonos.limpia()
except:
    self.conexión.rollback()
    raise
finally:
    cur.close()
def borra(self, registro):
    try:
        cur = self.conexión.cursor()
        cur.execute("delete from teléfonos where id_nombre = ?",
                    (registro.nombre.id,))
        cur.execute("delete from nombres where id = ?",
                    (registro.nombre.id,))
        self.conexión.commit()
    except:
        self.conexión.rollback()
        raise
    finally:
        cur.close()

```

La lista 11.30 presenta la clase **DBAgenda**, que sustituirá la clase **Agenda**. La clase **DBAgenda** mantiene el banco de datos en sincronía con las clases y objetos en la memoria, siendo responsable por todas las operaciones con el banco. Ese tipo de construcción en capas evita tener que escribir el código de manipulación y creación del banco en la clase **AppAgenda**.

Lo primero que hace la clase **DBAgenda** es verificar si el banco de datos ya existe. Esa verificación es hecha antes del pedido de conexión en el método `__init__`. Para saber si el banco ya existe, utilizamos la función `os.path.isfile(banco)`. En caso que el archivo no exista, será creado por el método `crea_banco`, llamado después de la conexión con el banco de datos. Vea que guardamos el objeto conexión como un atributo de **DBAgenda**.

El método `crea_banco` es muy simple, utilizando el método `executescript` de la conexión. Ese método ejecuta varios comandos de una sola vez. Para simplificar la creación del banco, escribimos el código que crea todas las tablas y llena la tabla tipos en la variable global **BANCO**. La ejecución de

varios comandos solo es posible porque están separados por ;.

El método **carga_tipos** realiza la lectura de todos los tipos de teléfono en el banco de datos y los guarda en la lista **tiposTeléfono**. Observe el cuidado en mantener los **id**. Ese valor será utilizado después para obtener el tipo correcto de cada teléfono.

El método **investigación_nombre** también tiene novedades. En él ejecutamos una consulta usando la función **count(*)**. Si un registro es encontrado, el método **carga_por_nombre** es llamado para transformar el resultado de nuestra consulta en una colección de objetos, del mismo modo que trabajamos con la agenda en el capítulo 10. Vea también que utilizamos la cláusula **where** para investigar en la tabla nombres directamente, puesto que no cargamos todos los registros del banco de datos en la memoria.

En **carga**, el resultado de nuestra consulta es transformado en un objeto **DBDatoAgenda**, después de crear una instancia de **DBNombre** con los campos **nombre** y **id** surgidos de nuestra consulta. Ese tipo de acceso es posible, pues registramos **self.conexión.row_factory = sqlite3.Row** en el método **__init__**. Una vez que el nombre fue cargado, utilizamos su **id** como **id_nombre** en nuestra próxima consulta, que cargará los valores de teléfono. Observe el detalle durante la lectura de los teléfonos y la transformación del resultado en **DBTeléfono**. Como el tipo aún no fue cargado y los tenemos a todos en la memoria, hacemos una investigación en **self.TiposTeléfono** para convertir el **id** en una instancia de **TipoTeléfono**. Es ese tipo de mapeo que no es tan simple de hacer y que es bastante trabajoso por la gran cantidad de código necesaria para mantenerlo. Es en ese momento que un ORM ayuda. Una vez que todo fue convertido, **nuevo** es retornado con todos los datos precargados.

El método **lista** ejecuta una consulta total de la tabla nombres. Sin embargo, para evitar la creación de una gran lista, utilizamos la instrucción **yield** de Python que retorna cada valor cargado, uno por vez. En realidad, el método **lista** retorna un generador (*generator*) que puede ser utilizado en un **for** de Python. Eso evita tener que cargar y convertir todos los valores antes de tener los primeros resultados en la pantalla.

En **nuevo**, convertimos un objeto del tipo **DBDatoAgenda** en registros de las tablas nombres y teléfonos. Esa conversión es realizada dentro de una transacción, de ahí el porqué de utilizar explícitamente un cursor (**cur**). Realizar esa operación dentro de una transacción permitirá mejorar la consistencia del banco de datos, pues si ocurre un error antes de completar todas las operaciones, el estado del banco de datos será revertido al estado anterior al comienzo de nuestras operaciones. Como el **id** de nombres es generado automáticamente, utilizamos la propiedad **lastrowid** de nuestro cursor, después de la ejecución del **insert**. De esta forma, podemos utilizar el nuevo id para llenar la tabla de teléfonos. Realizamos entonces el mismo proceso con cada nuevo teléfono, atribuyendo el valor de **lastrowid** al **id** del teléfono.

El método **actualiza** es más complejo. Durante la actualización de un registro necesitamos actualizar el campo **nombre** en la tabla de nombres. En nuestra solución casera, no tenemos manera de saber si el nombre fue alterado o no. Podríamos alterar la clase para saber cuándo el nombre fue alterado y ejecutar el **update** solo cuando sea necesario. Esta es otra característica de las bibliotecas de ORM que traen ese tipo de funcionalidad en sus clases. Para mantener la agenda lo más simple posible, **nombre** siempre será actualizado. Para cada **teléfono**, hacemos la verificación del valor de **teléfono.id**. Si un teléfono ya tiene un valor en **id**, probablemente ya está registrado en el banco de

datos y necesita ser actualizado. En caso que aún no posea un valor en **id**, probablemente se trata de un teléfono introducido durante la alteración. De esa forma, elegimos entre hacer un **insert** o un **update** en la tabla teléfonos. Por último, verificamos si la lista de borrados tiene una lista de **id** a borrar. Esa lista fue construida por la clase **DBListaÚnica**, donde guardamos el **id** de cada elemento borrado. Ese cambio fue necesario debido a la diferencia de los nuevos registros y de los registros alterados; los registros borrados son removidos de nuestra lista. Si no mantenemos la lista de los **id** borrados, no sabremos qué teléfonos fueron removidos, y nuestro banco se volvería inconsistente. Para cada **id** en la lista de borrados, ejecutamos un **delete**. Luego terminamos a transacción con un **commit** para asegurarnos que todas las operaciones fueron registradas en el banco de datos, y solo entonces borramos la lista de teléfonos removidos con el método **limpia**.

El método **borra** es un poco más simple. En él utilizamos la misma estructura de protección y una transacción. Lo interesante es que primero borramos los teléfonos y, después, los nombres. Dada la forma en que generamos nuestro banco de datos ese orden no importa, porque no estamos utilizando los recursos de integridad referencial del banco.

El programa de la agenda es presentado por entero en la lista 11.31.

► Lista 11.31 – Agenda con banco de datos completo

```
import sys
import sqlite3
import os.path
from functools import total_ordering

BANCO = """
create table tipos(id integer primary key autoincrement,
    descripción text);
create table nombres(id integer primary key autoincrement,
    nombre text);
create table teléfonos(id integer primary key autoincrement,
    id_nombre integer,
    número text,
    id_tipo integer);
insert into tipos(descripción) values ("Celular");
insert into tipos(descripción) values ("Fijo");
insert into tipos(descripción) values ("Fax");
insert into tipos(descripción) values ("Casa");
insert into tipos(descripción) values ("Trabajo");
"""

def nulo_o_vacio(texto):
    return texto == None or not texto.strip()

def valida_franja_entero(pregunta, inicio, fin, estándar=None):
    while True:
```

```

try:
    entrada = input(pregunta)
    if nulo_o_vacio(entrada) and estándar != None:
        entrada = estándar
    valor = int(entrada)
    if inicio <= valor <= fin:
        return(valor)
except ValueError:
    print("Valor inválido, favor digitar entre %d y %d" %
          (inicio, fin))

```

```

def valida_franja_entero_o_blanco(pregunta, inicio, fin):
    while True:
        try:
            entrada = input(pregunta)
            if nulo_o_vacio(entrada):
                return None
            valor = int(entrada)
            if inicio <= valor <= fin:
                return(valor)
        except ValueError:
            print("Valor inválido, favor digitar entre %d y %d" %
                  (inicio, fin))

```

```

class ListaÚnica:
    def __init__(self, elem_class):
        self.lista = []
        self.elem_class = elem_class
    def __len__(self):
        return len(self.lista)
    def __iter__(self):
        return iter(self.lista)
    def __getitem__(self, p):
        return self.lista[p]
    def indiceVálido(self, i):
        return i>=0 and i<len(self.lista)
    def adiciona(self, elem):
        if self.investigación(elem) == -1:
            self.lista.append(elem)
    def remove(self, elem):
        self.lista.remove(elem)

```



```

def investigación(self, elem):
    self.verifica_tipo(elem)
    try:
        return self.lista.index(elem)
    except ValueError:
        return -1

def verifica_tipo(self, elem):
    if type(elem)!=self.elem_class:
        raise TypeError("Tipo inválido")

def ordena(self, clave=None):
    self.lista.sort(key=clave)

```

```

class DBListaÚnica(ListaÚnica):
    def __init__(self, elem_class):
        super().__init__(elem_class)
        self.borrados = []

    def remove(self, elem):
        if elem.id is not None:
            self.borrados.append(elem.id)
            super().remove(elem)

    def limpia(self):
        self.borrados = []

```

@total_ordering

```

class Nombre:
    def __init__(self, nombre):
        self.nombre = nombre

    def __str__(self):
        return self.nombre

    def __repr__(self):
        return "<Clase {3} en 0x{0:x} Nombre: {1} Clave: {2}>".format(
            id(self), self.__nombre, self.__clave,
            type(self).__name__)

    def __eq__(self, otro):
        return self.nombre == otro.nombre

    def __lt__(self, otro):
        return self.nombre < otro.nombre

```

@property

```

def nombre(self):
    return self.__nombre

```

```
@nombre.setter
def nombre(self, valor):
    if nulo_o_vacio(valor):
        raise ValueError("Nombre no puede ser nulo ni en blanco")
    self.__nombre = valor
    self.__clave = Nombre.CreaClave(valor)
```

```
@property
```

```
def clave(self):
    return self.__clave
```

```
@staticmethod
```

```
def CreaClave(nombre):
    return nombre.strip().lower()
```

```
class DBNombre(Nombre):
```

```
def __init__(self, nombre, id_=None):
    super().__init__(nombre)
    self.id = id_
```

```
@total_ordering
```

```
class TipoTeléfono:
```

```
def __init__(self, tipo):
    self.tipo = tipo
def __str__(self):
    return "({0})".format(self.tipo)
def __eq__(self, otro):
    if otro is None:
        return False
    return self.tipo == otro.tipo
```

```
def __lt__(self, otro):
    return self.tipo < otro.tipo
```

```
class DBTipoTeléfono(TipoTeléfono):
```

```
def __init__(self, id_, tipo):
    super().__init__(tipo)
    self.id = id_
```

```
class Teléfono:
```

```
def __init__(self, número, tipo=None):
    self.número = número
    self.tipo = tipo
def __str__(self):
    if self.tipo!=None:
        tipo = self.tipo
```

```

else:
    tipo = ""
    return "{0} {1}".format(self.número, tipo)
def __eq__(self, otro):
    return self.número == otro.número and (
        (self.tipo == otro.tipo) or (
            self.tipo == None or otro.tipo == None))
@property
def número(self):
    return self.__número
@número.setter
def número(self, valor):
    if nulo_o_vacio(valor):
        raise ValueError("Número no puede ser None o en blanco")
    self.__número = valor
class DBTeléfono(Teléfono):
    def __init__(self, número, tipo=None, id_=None, id_nombre=None):
        super().__init__(número, tipo)
        self.id = id_
        self.id_nombre = id_nombre
class DBTeléfonos(DBListaÚnica):
    def __init__(self):
        super().__init__(DBTeléfono)
class DBTiposTeléfono(ListaÚnica):
    def __init__(self):
        super().__init__(DBTipoTeléfono)
class DBDatoAgenda:
    def __init__(self, nombre):
        self.nombre = nombre
        self.teléfonos = DBTeléfonos()
@property
def nombre(self):
    return self.__nombre
@nombre.setter
def nombre(self, valor):
    if type(valor) != DBNombre:
        raise TypeError("nombre debe ser una instancia de la clase DBNombre")
    self.__nombre = valor

```

```

def investigaciónTeléfono(self, teléfono):
    posición = self.teléfonos.investigación(DBTeléfono(teléfono))
    if posición == -1:
        return None
    else:
        return self.teléfonos[posición]

class DBAgenda:
    def __init__(self, banco):
        self.tiposTeléfono = DBTiposTeléfono()
        self.banco = banco
        nuevo = not os.path.isfile(banco)
        self.conexión = sqlite3.connect(banco)
        self.conexión.row_factory = sqlite3.Row
        if nuevo:
            self.crea_banco()
            self.cargaTipos()
    def cargaTipos(self):
        for tipo in self.conexión.execute("select * from tipos"):
            id_ = tipo["id"]
            descripción = tipo["descripción"]
            self.tiposTeléfono.adiciona(DBTipoTeléfono(id_, descripción))
    def crea_banco(self):
        self.conexión.executescript(BANCO)
    def investigaciónNombre(self, nombre):
        if not isinstance(nombre, DBNombre):
            raise TypeError("nombre debe ser del tipo DBNombre")
        encontrado = self.conexión.execute("""select count(*)
                                             from nombres where nombre = ?""",
                                             (nombre.nombre,)).fetchone()

        if(encontrado[0]>0):
            return self.carga_por_nombre(nombre)
        else:
            return None
    def carga_por_id(self, id):
        consulta = self.conexión.execute(
            "select * from nombres where id = ?", (id,))
        return carga(consulta.fetchone())
    def carga_por_nombre(self, nombre):
        consulta = self.conexión.execute(

```

```

        "select * from nombres where nombre = ?", (nombre.nombre,))
    return self.carga(consulta.fetchone())
def carga(self, consulta):
    if consulta is None:
        return None
    nuevo = DBDatoAgenda(DBNombre(consulta["nombre"], consulta["id"]))
    for teléfono in self.conexión.execute(
        "select * from teléfonos where id_nombre = ?",
        (nuevo.nombre.id,)):
        ntel = DBTeléfono(teléfono["número"], None,
                           teléfono["id"], teléfono["id_nombre"])
    for tipo in self.tiposTeléfono:
        if tipo.id == teléfono["id_tipo"]:
            ntel.tipo = tipo
            break
    nuevo.teléfonos.adiciona(ntel)
    return nuevo
def lista(self):
    consulta = self.conexión.execute(
        "select * from nombres order by nombre")
    for registro in consulta:
        yield self.carga(registro)
def nuevo(self, registro):
    try:
        cur = self.conexión.cursor()
        cur.execute("insert into nombres(nombre) values (?)",
                    (str(registro.nombre),))
        registro.nombre.id = cur.lastrowid
        for teléfono in registro.teléfonos:
            cur.execute("""insert into teléfonos(número,
                    id_tipo, id_nombre) values (?,?,""",
                    (teléfono.número, teléfono.tipo.id,
                     registro.nombre.id))
            teléfono.id = cur.lastrowid
        self.conexión.commit()
    except:
        self.conexión.rollback()
        raise

```

finally:

cur.close()

def actualiza(**self**, registro):

try:

cur = **self**.conexión.cursor()

cur.execute("update nombres set nombre=? where id = ?",
 (str(registro.nombre), registro.nombre.id))

for teléfono **in** registro.teléfonos:

if teléfono.id **is** **None**:

cur.execute("""insert into teléfonos(número,
 id_tipo, id_nombre)
 values (?, ?, ?)""",
 (teléfono.número, teléfono.tipo.id,
 registro.nombre.id))

teléfono.id = cur.lastrowid

else:

cur.execute("""update teléfonos set número=?,
 id_tipo=?, id_nombre=?
 where id = ?""",
 (teléfono.número, teléfono.tipo.id,
 registro.nombre.id, teléfono.id))

for borrado **in** registro.teléfonos.borrados:

cur.execute("delete from teléfonos where id = ?",
 (borrado,))

self.conexión.commit()

registro.teléfonos.limpia()

except:

self.conexión.rollback()

raise

finally:

cur.close()

def borra(**self**, registro):

try:

cur = **self**.conexión.cursor()

cur.execute("delete from teléfonos where id_nombre = ?",
 (registro.nombre.id,))

cur.execute("delete from nombres where id = ?",
 (registro.nombre.id,))

```

        self.conexión.commit()
    except:
        self.conexión.rollback()
        raise
    finally:
        cur.close()

```

class Menú:

```

def __init__(self):
    self.opciones = [["Salir", None]]
def adicionaopción(self, nombre, función):
    self.opciones.append([nombre, función])
def exhibe(self):
    print("====")
    print("Menú")
    print("====\n")
    for i, opción in enumerate(self.opciones):
        print("[{0}] - {1}".format(i, opción[0]))
    print()
def ejecute(self):
    while True:
        self.exhibe()
        elija = valida_franja_entero("Elija una opción: ",
                                     0, len(self.opciones)-1)
        if elija == 0:
            break
        self.opciones[elija][1]()

```

class AppAgenda:

```

@staticmethod
def pide_nombre():
    return(input("Nombre: "))
@staticmethod
def pide_teléfono():
    return(input("Teléfono: "))
@staticmethod
def muestra_datos(datos):
    print("Nombre: %s" % datos.nombre)
    for teléfono in datos.teléfonos:
        print("Teléfono: %s" % teléfono)
    print()

```

```

@staticmethod
def muestra_datos_teléfono(datos):
    print("Nombre: %s" % datos.nombre)
    for i, teléfono in enumerate(datos.teléfonos):
        print("{0} - Teléfono: {1}".format(i, teléfono))
    print()
def __init__(self, banco):
    self.agenda = DBAgenda(banco)
    self.menú = Menú()
    self.menú.adicionaopción("Nuevo", self.nuevo)
    self.menú.adicionaopción("Alterar", self.alterar)
    self.menú.adicionaopción("Borra", self.borra)
    self.menú.adicionaopción("Lista", self.lista)
    self.ultimo_nombre = None
def pide_tipo_teléfono(self, estándar=None):
    for i, tipo in enumerate(self.agenda.tiposTeléfono):
        print(" {0} - {1} ".format(i, tipo), end=None)
        t = valida_franja_entero("Tipo: ", 0,
                                len(self.agenda.tiposTeléfono)-1, estándar)
    return self.agenda.tiposTeléfono[t]
def investigación(self, nombre):
    if type(nombre)==str:
        nombre = DBNombre(nombre)
    dato = self.agenda.investigaciónNombre(nombre)
    return dato
def nuevo(self):
    nuevo = AppAgenda.pide_nombre()
    if nulo_o_vacío(nuevo):
        return
    nombre = DBNombre(nuevo)
    if self.investigación(nombre) != None:
        print(";Nombre ya existe!")
    return
    registro = DBDatoAgenda(nombre)
    self.menú_teléfonos(registro)
    self.agenda.nuevo(registro)
def borra(self):
    nombre = AppAgenda.pide_nombre()

```



```

if(nulo_o_vacíó(nombre)):
    return
p = self.investigación(nombre)
if p != None:
    self.agenda.borra(p)
else:
    print("Nombre no encontrado.")
def altera(self):
    nombre = AppAgenda.pide_nombre()
    if(nulo_o_vacíó(nombre)):
        return
    p = self.investigación(nombre)
    if p != None:
        print("\\nEncontrado:\\n")
        AppAgenda.muestra_datos(p)
        print("Digite enter en caso que no quiera alterar el nombre")
        nuevo = AppAgenda.pide_nombre()
        if not nulo_o_vacíó(nuevo):
            p.nombre.nombre = nuevo
        self.menú_teléfonos(p)
        self.agenda.actualiza(p)
    else:
        print(";Nombre no encontrado!")
def menú_teléfonos(self, datos):
    while True:
        print("\\nEditando teléfonos\\n")
        AppAgenda.muestra_datos_teléfono(datos)
        if(len(datos.teléfonos)>0):
            print("\\n[A] - alterar\\n[D] - borrar\\n", end="")
            print("[N] - nuevo\\n[S] - salir\\n")
            operación = input("Elija una operación: ")
            operación = operación.lower()
            if operación not in ["a","d","n", "s"]:
                print("Operación inválida. Digite A, D, N o S")
                continue
            if operación == 'a' and len(datos.teléfonos)>0:
                self.altera_teléfonos(datos)
            elif operación == 'd' and len(datos.teléfonos)>0:
                self.borra_teléfono(datos)

```

```

        elif operación == 'n':
            self.nuevo_teléfono(datos)
        elif operación == "s":
            break
def nuevo_teléfono(self, datos):
    teléfono = AppAgenda.pide_teléfono()
    if nulo_o_vacío(teléfono):
        return
    if datos.investigaciónTeléfono(teléfono) != None:
        print("Teléfono ya existe")
    tipo = self.pide_tipo_teléfono()
    datos.teléfonos.adiciona(DBTeléfono(teléfono, tipo))
def borra_teléfono(self, datos):
    t = valida_franja_entero_o_blanco(
        "Digite la posición del número a borrar, enter para salir: ",
        0, len(datos.teléfonos)-1)
    if t == None:
        return
    datos.teléfonos.remove(datos.teléfonos[t])
def altera_teléfonos(self, datos):
    t = valida_franja_entero_o_blanco(
        "Digite la posición del número a alterar, enter para salir: ",
        0, len(datos.teléfonos)-1)
    if t == None:
        return
    teléfono = datos.teléfonos[t]
    print("Teléfono: %s" % teléfono)
    print("Digite enter en caso que no quiera alterar el número")
    nuevoteléfono = AppAgenda.pide_teléfono()
    if not nulo_o_vacío(nuevoteléfono):
        teléfono.número = nuevoteléfono
    print("Digite enter en caso que no quiera alterar el tipo")
    teléfono.tipo = self.pide_tipo_teléfono(
        self.agenda.tiposTeléfono.investigación(teléfono.tipo))
def lista(self):
    print("\nAgenda")
    print("-"*60)
    for e in self.agenda.lista():

```

```

        AppAgenda.muestra_datos(e)
    print("-"*60)
    def ejecute(self):
        self.menú.ejecute()
if __name__ == "__main__":
    if len(sys.argv) > 1:
        app = AppAgenda(sys.argv[1])
        app.ejecute()
    else:
        print("Error: nombre del banco de datos no informado")
        print("          agenda.py nombre_del_banco")

```

En la clase **AppAgenda**, modificamos las opciones del menú y los tipos usados en las investigaciones. Observe que con la utilización de la clase **DBAgenda**, conseguimos aislar las operaciones de banco de datos de la clase **AppAgenda**. Como no tenemos operaciones de lectura y grabación, el nombre del banco de datos debe ser pasado obligatoriamente en la línea de comando. Un mensaje de error será exhibido en caso que usted se olvide de ese detalle.

Próximos pasos

La programación es un largo camino que usted empezó a recorrer. En este libro, mostramos las técnicas básicas para que continúe sus estudios. La programación de computadoras es un área de conocimiento muy grande y rica. Este libro presentó una introducción a la programación utilizando el lenguaje Python. Los próximos pasos dependen de su área de interés. Usted no necesita estudiar todos los tópicos o aun seguir el orden en que son presentados en este capítulo. Vamos a listar algunos tópicos.

12.1 Programación funcional

La programación funcional es un paradigma de programación diferente del que aprendimos aquí con Python. Aunque Python tenga recursos de un lenguaje funcional, permitiendo mezclar los dos paradigmas en un solo programa. Para enriquecer su experiencia en programación, estudiar un lenguaje puramente funcional, como LISP, Scheme o F#, es realmente interesante, siendo un área que debería popularizarse cada vez más. Estudiar una nueva forma de organizar sus programas lo ayudará a entender mejor varios conceptos que ya aprendió y a modificar la forma en que encara la programación. Actualmente, diversos lenguajes son llamados híbridos o multiparadigmas, pues mezclan o posibilitan la utilización de recursos de programación funcional e imperativa. Python es uno de ellos. Además de eso, algunas propiedades de la programación funcional son excelentes para resolver o abordar problemas de competencia, cada vez más comunes hoy en día.

El libro SICP – Structure and Interpretation of Computer Programs, de Harold Abelson y Gerald Jay Sussman, puede ser leído online en el sitio <http://mitpress.mit.edu/sicp/full-text/book/book.html>. El libro es usado en diversos cursos de computación y es considerado uno de los mejores disponibles.

12.2 Algoritmos

Hasta ahora utilizamos lo básico de la programación para resolver nuestros problemas. A medida que vaya ganando experiencia, tendrá que estudiar nuevos algoritmos y entender algoritmos que ya utiliza hoy, sin ser conciente de ello. Un ejemplo es la comprensión de técnicas de dispersión (*hashes*), que utilizamos en Python con diccionarios, o cómo funcionan internamente las listas. Cuanto más grandes sean sus programas, más importante será entender cómo funciona cada algoritmo, conocer sus aplicaciones y limitaciones.

El lenguaje C también es recomendado para el estudio de algoritmos y estructuras de datos. Al contrario de Python, en C usted está solo y él viene casi sin baterías. La ventaja de estudiar y aprender a programar en C es conocer los detalles de cada implementación y controlar cada paso de

sus programas.

Un libro recomendado para estudios profundos es “Algoritmos: teoría y práctica”, de Thomas H. Cormen. Aunque la lectura no sea de las más fáciles, su contenido es esclarecedor. No lea ese libro antes de programar por lo menos en dos lenguajes de programación. Su lectura está recomendada para personas que deseen realmente profundizar en el asunto, y que normalmente están cursando ciencias de la computación.

12.3 Juegos

La programación de juegos es una de las áreas que más recompensa e inspira. Además de eso, lo que aprende programando juegos sirve para solucionar diversos problemas del día a día. Python presenta diversas bibliotecas externas prontas para la creación de juegos en 2D o 3D. Esas bibliotecas incluyen la manipulación de imágenes, sonidos, control del tiempo y aun de archivos.

Pygame (<http://www.pygame.org>): biblioteca multimedia fácil de aprender. Permite la manipulación de superficies de diseño y la creación de juegos con gráficos en 2D y sonido. Si tiene curiosidad por saber cómo crear juegos simples, visite el proyecto Invasores (<http://www.sourceforge.net/projects/invasores>). El proyecto fue enteramente desarrollado en Python y es open source. Puede bajar y estudiar el código-fuente, hacer modificaciones y probar. Todo está escrito en Python.

Panda 3D (<http://www.panda3d.org/>): biblioteca gráfica, desarrollada por los estudios Disney, open source y supercompleta. Con Panda 3D puede crear juegos profesionales y, lo mejor de todo, usando Python. La biblioteca es poderosa y compleja, y ha sido utilizada en juegos comerciales. No se olvide de revisar álgebra lineal y estudiar gráficos tridimensionales antes de aventurarse en la documentación (está en inglés).

PyOgre (<http://www.ogre3d.org/tikiwiki/PyOgre>): otra biblioteca o kit de desarrollo de juegos 3D. La PyOgre es también poderosa y compleja.

Si empieza a estudiar esas bibliotecas, encare esa tarea como un proyecto de por lo menos un año. Esas bibliotecas son grandes y permiten la creación de aplicaciones multimedia, además de juegos, claro. Lo ideal es realizar ese estudio en grupo: visite el sitio <http://www.unidev.com.br>.

12.4 Orientación a objetos

En el capítulo 10, solo hicimos una introducción a la programación orientada a objetos. Como ya fue dicho, ese es un asunto realmente extenso que vale la pena ser estudiado. Python es un lenguaje que implementa casi todos los conceptos de orientación a objeto, pero usted debe aprender a dominarlos. Estudie poliformismo, interfaces y estándares de proyecto (*design patterns*).

Un libro que ayuda a descifrar la orientación a objetos es “Head First Design Patterns”, de Eric T. Freeman, Elisabeth Robson, Bert Bates y Kathy Sierra. El libro presenta los conceptos básicos y los principales estándares del proyecto de forma simple y divertida, con diversas historias y gráficos.

Un libro clásico sobre estándares de proyecto es “*Design Patterns: Elements of Reusable Object-Oriented Software*”, de Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. El libro presenta un contenido denso en formato de referencia. No es indicado para una introducción a

estándares de proyecto, pero algún día usted debe leerlo.

12.5 Banco de datos

Aunque hayamos abordado banco de datos en el capítulo 11, el asunto es extenso.

Después de leer la documentación del módulo `sqlite3`, puede aventurarse con bibliotecas más modernas que aíslan sus programas del banco de datos en sí, volviendo la manipulación de sus datos casi transparente. Un consejo es la biblioteca `SQLAlchemy` (<http://www.sqlalchemy.org/>).

12.6 Sistemas web

Uno de los asuntos más vigentes hoy en día es la programación de sistemas Web. Esos sistemas también pueden ser escritos en Python. Mientras tanto, la comunicación de un programa con el servidor de páginas web está repleta de detalles y vuelve sus programas repetitivos y trabajosos de escribir. Si usted desea aprender a escribir, o se interesa por sistemas web, estudie el Django (<http://www.djangoproject.com/>). El Django estructura sus sistemas para que se vuelvan fáciles de escribir y modificar. Además de eso, las bibliotecas del Django ya realizan las tareas repetitivas de comunicación con el servidor web y aun con el banco de datos.

12.7 Otras bibliotecas Python

Además que Python viene con “baterías incluidas”, usted encuentra en Internet diversas bibliotecas o más baterías para hacer prácticamente todo. Un buen sitio para buscar un módulo es Python Package Index – PyPI (<http://pypi.python.org/pypi>).

Para aplicaciones científicas o de cálculo intenso, no se olvide de ver los módulos NumPy (<http://numpy.scipy.org/>) y Scipy (<http://www.scipy.org/>).

Si necesita trabajar con gráficos (columna, barra, torta, superficie y muchos otros), busque el módulo `matplotlib` (<http://matplotlib.sourceforge.net/>).

Otro proyecto que no puede dejar de ser visitado es el Pycairo (<http://cairographics.org/pycairo/>), que permite el diseño de gráficos con recursos avanzados, como suavizado de curvas y transparencias. Para trabajar con imágenes (PNG, JPG etc.), el módulo PIL, o *Python Image Library* (<http://www.pythonware.com/products/pil/>), es referencia.

12.8 Listas de discusión

Una forma de no quedar aislado es participar de listas de discusión. Para profundizar y continuar estudiando Python, inscríbase en la lista <http://www.es.python.org/> (<https://mail.python.org/mailman/listinfo/python-es>) o la lista <http://listas.python.org.ar/listinfo/pyar>. Esa lista es muy activa y cuenta con una comunidad participativa que recibe dudas básicas y avanzadas. Antes de preguntar, lea las guías disponibles en el sitio y las reglas de cómo hacer preguntas. El histórico de la lista también trae miles de preguntas y respuestas que puede investigar. La wiki es rica en ejemplos de problemas y soluciones escritas en Python y disponibles en español.

Ya para quien está estudiando Django, la lista de discusión es <http://django.es/>.

Mensajes de error

Errores siempre pueden ocurrir. A veces digitamos algo equivocado, o nos olvidamos de digitar algo importante.

El intérprete de Python informa de los errores por medio de mensajes que indican el tipo de error, así como dónde ocurrió (archivo y línea). Las secciones de este apéndice muestran la lista de un programa y el mensaje de error respectivo. Recuerde que el intérprete sigue reglas como cualquier programa y que, a veces, puede haber mensajes causados por errores en otras líneas de su programa. Utilice los mensajes de error como un buen dato de lo que ocurrió. Lo que ellas realmente indican es dónde, en su programa, el intérprete fue interrumpido y la causa de esa interrupción. Es investigando ese dato que usted encontrará el verdadero error.

A.1 SyntaxError

Uno de los tipos más comunes. Un error de sintaxis sucede cuando el intérprete no logra leer lo que usted escribió. En otras palabras, su programa está mal formado, normalmente con errores de digitación o símbolos olvidados.

```
a="5
File "errores/sintax.py", line 1
    a="5
    ^
SyntaxError: EOL while scanning string literal
```

En el ejemplo anterior, la línea fue terminada sin cerrar comillas.

```
for e in [1,2,3]
print e
File "errores/sintax2.py", line 1
for e in [1,2,3]
    ^
SyntaxError: invalid syntax
```

Todas las líneas con **for**, **while**, **if** y **else** deben ser cerradas con el símbolo de **:** para indicar el comienzo de un nuevo bloque.

```
a=10
print a
```

```
File "errores/sintax4.py", line 2
```

```
print a
```

```
^
```

```
SyntaxError: invalid syntax
```

Aquí la función `print` fue utilizada, pero observe que el parámetro `a` no fue escrito entre paréntesis. Observe que un acento circunflejo es exhibido en la columna en que el intérprete considera que el error puede haber acontecido. Esa indicación es solo un consejo, debiendo ser investigada caso por caso. Algunos errores pueden hacer que el intérprete se pierda, indicando el lugar equivocado. Siempre lea la línea indicada en el mensaje de error, pero no se olvide de mirar también las líneas anteriores, en caso que no encuentre el error.

Siempre que ocurre un error de sintaxis, verifique:

1. La línea donde ocurrió un error (*line*).
2. Si usted cerró todas las comillas que abrió, lo mismo vale para paréntesis.
3. Los dos puntos después de `while`, `for`, `if`, `else`, definiciones de función, métodos y clases.
4. Si usted no cambió letras minúsculas por mayúsculas y viceversa.
5. Si usted digitó correctamente todos los nombres.

A.2 IndentationError

```
x=0
```

```
while x<10:
```

```
print(x)
```

```
    x=x+1
```

```
File "errores/sintax5.py", line 4
```

```
    x=x+1
```

```
^
```

```
IndentationError: unindent does not match any outer indentation level
```

En ese caso, observe que la línea `x=x+1` no está alineada ni con el bloque del `print` ni con el `while`. Todas las líneas de un mismo bloque deben estar alineadas en la misma columna.

Una variación muy difícil de percibir de ese error es cuando mezclamos espacios en blanco con tabulaciones (tabs). Configure su editor de textos para sustituir tabs por espacios en blanco o viceversa. Jamás mezcle tabulaciones y espacios en blanco en sus programas en Python.

A.3 KeyError

```
>>> mensualidades = {"auto":500, "casa":1500}
```

```
>>> print(mensualidades["seguro"])
```

```
Traceback (most recent call last):
```



```
File "<pyshell#54>", line 1, in <module>
print(mensualidades["seguro"])
KeyError: 'seguro'
```

La excepción **KeyError** ocurre cuando accedemos o tratamos de acceder a un diccionario usando una clave que no existe. En el ejemplo anterior, el diccionario **mensualidades** contiene las claves **auto** y **casa**. En la línea de la función **print**, tratamos de acceder a **mensualidades["seguro"]**. En ese caso, **"seguro"** es la clave que causa el mensaje de error, puesto que la misma no pertenece al diccionario. Atención al trabajar con claves del tipo cadena de caracteres, pues Python diferencia letras minúsculas de mayúsculas.

A.4 NameError

```
while x<0:
print(x)
    x=x+1
```

```
Traceback (most recent call last):
  File "errores/sintax3.py", line 1, in <module>
while x<0:
NameError: name 'x' is not defined
```

Aquí la variable **x** es utilizada en **while**, aún antes de ser inicializada. Toda variable en Python necesita ser inicializada antes de ser utilizada. Recuerde que para inicializar una variable debemos atribuirle un valor inicial. En este caso, podríamos escribir **x=0** antes de la línea de **while** para inicializar la variable **x** con cero, resolviendo el error.

Al recibir ese error, verifique también si escribió correctamente el nombre de la variable. Recuerde que el nombre de una variable, como cualquier identificador en Python, tiene en cuenta variaciones como letras minúsculas y mayúsculas. Por ejemplo **X=0** no resolvería el error, pues **x** y **X** son variables diferentes. No se olvide también de digitar correctamente los acentos, pues nombres acentuados son también diferentes de nombres sin acentos. Así, **posicion** es diferente de **posición**.

A.5 ValueError

La excepción **ValueError** puede ocurrir por diversas causas. Una de ellas es la imposibilidad de convertir un valor con las funciones **int** o **float**:

```
>>> int("abc")
Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    int("abc")
ValueError: invalid literal for int() with base 10: 'abc'
>>> float("maría")
Traceback (most recent call last):
```

```
File "<pyshell#68>", line 1, in <module>
```

```
float("maría")
```

```
ValueError: could not convert string to float: maría
```

Este error puede ocurrir si, por ejemplo, el valor retornado por la función **input** es inválido.

La excepción también ocurre cuando buscamos una cadena de caracteres que no existe, como se muestra a continuación:

```
>>> s="Hola mundo"
```

```
>>> s.index("rei")
```

```
Traceback (most recent call last):
```

```
File "<pyshell#64>", line 1, in <module>
```

```
s.index("rei")
```

```
ValueError: substring not found
```

A.6 TypeError

Esta excepción ocurre si tratamos de llamar una función usando más parámetros de los que recibe. En el ejemplo de abajo, la función **float** fue llamada con dos parámetros: 35 y 4. Recuerde que los números en Python deben ser escritos en el formato americano, separando la parte entera de la parte decimal de un número con punto y no con coma. Ese también es un ejemplo que un error puede ser presentado como otro error, exigiendo que usted siempre interprete el mensaje de modo de saber lo que realmente sucedió y no de manera literal.

```
>>> float(35,4)
```

```
Traceback (most recent call last):
```

```
File "<pyshell#69>", line 1, in <module>
```

```
float(35,4)
```

```
TypeError: float() takes at most 1 argument (2 given)
```

TypeError también ocurre cuando cambiamos el tipo de un índice. En el ejemplo de abajo, tratamos de utilizar la cadena de caracteres “marrón” como índice de una lista. Recuerde que las listas solo pueden ser indexadas por números enteros. Los diccionarios, a su vez, permiten índices del tipo cadena de caracteres.

```
>>> s["amarillo", "rojo", "verde"]
```

```
>>> s["marrón"]
```

```
Traceback (most recent call last):
```

```
File "<pyshell#71>", line 1, in <module>
```

```
s["marrón"]
```

```
TypeError: string indices must be integers
```

A.7 IndexError

IndexError indica que fue utilizado un valor inválido de índice. En el ejemplo a continuación, la

cadena de caracteres `s` contiene solo cinco elementos, pudiendo tener índices de 0 a 4.

```
>>> s="ABCDE"
```

```
>>> s[10]
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#66>", line 1, in <module>
```

```
    s[10]
```

```
IndexError: string index out of range
```

Referencias

- Cormen, Thomas H; Leiserson, Charles E; Rivest, Ronald L; Stein, Clifford. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2002.
- Pilgrim, M. *Dive into Python 3*. 2 ed. Apress, 2009.
- Python Foundation. Python 3.1.2 Tutorial. 2010 Disponible en <http://docs.python.org/py3k/tutorial>.
- Summerfield, M. *Programming in Python 3 – A Complete Introduction to the Python Language*. 2 ed. Addison-Wesley, 2010.