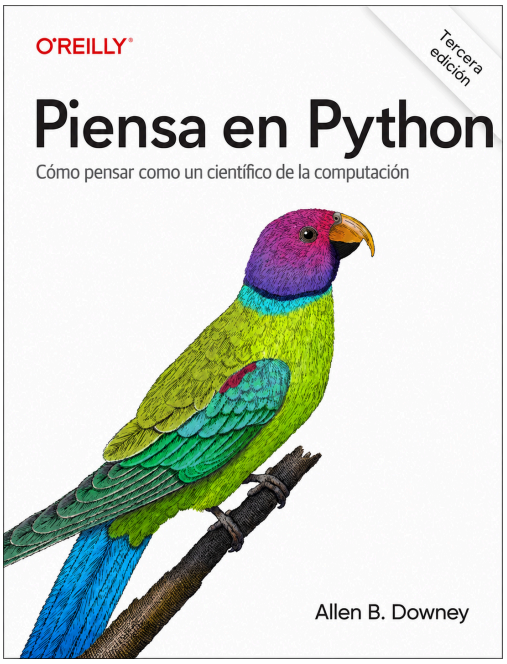
Think Python en español

Think Python es una introducción a Python para personas que nunca han programado, o para quienes lo han intentado y se han encontrado con dificultades.

Esta edición en español traduce la tercera edición de *Think Python: How to Think Like a Computer Scientist*, de Allen B. Downey, con traducción de midudev.



Puedes pedir las versiones impresa y electrónica de *Think Python 3e* en Bookshop.org y Amazon.

Aquí está la página principal del libro en Green Tea Press.

Descargar el libro completo en PDF

Los notebooks

Capítulo 1: Programar como una forma de pensar

- Abrir el capítulo 1 en Colab

Capítulo 2: Variables y sentencias

- Abrir el capítulo 2 en Colab

Capítulo 3: Funciones

- Abrir el capítulo 3 en Colab

Capítulo 4: Funciones e interfaces

- Abrir el capítulo 4 en Colab

Capítulo 5: Condicionales y recursión

- Abrir el capítulo 5 en Colab

Capítulo 6: Valores de retorno

- Abrir el capítulo 6 en Colab

Capítulo 7: Iteración y búsqueda

- Abrir el capítulo 7 en Colab

Capítulo 8: Cadenas y expresiones regulares

- Abrir el capítulo 8 en Colab

Capítulo 9: Listas

- Abrir el capítulo 9 en Colab

Capítulo 10: Diccionarios

- Abrir el capítulo 10 en Colab

Capítulo 11: Tuplas

- Abrir el capítulo 11 en Colab

Capítulo 12: Análisis y generación de texto

- Abrir el capítulo 12 en Colab

Capítulo 13: Archivos y bases de datos

- Abrir el capítulo 13 en Colab

Capítulo 14: Clases y funciones

- Abrir el capítulo 14 en Colab

Capítulo 15: Clases y métodos

- Abrir el capítulo 15 en Colab

Capítulo 16: Clases y objetos

- Abrir el capítulo 16 en Colab

Capítulo 17: Herencia

- Abrir el capítulo 17 en Colab

Capítulo 18: Extras de Python

- Abrir el capítulo 18 en Colab

Capítulo 19: Reflexiones finales

- Abrir el capítulo 19 en Colab

Recursos para docentes

Cada capítulo también tiene un notebook en blanco con el texto original y la mayor parte del código eliminado. Estos notebooks sirven para hacer ejercicios guiados donde el alumnado completa los huecos.

Aquí están los enlaces a los notebooks en blanco.

Puedes pedir las versiones impresa y ebook de *Think Python 3e* en Bookshop.org y Amazon.

Prefacio

¿Para quién es este libro?

Si quieres aprender a programar, has llegado al lugar adecuado. Python es uno de los mejores lenguajes de programación para principiantes, y también una de las habilidades más demandadas.

También has llegado en el momento adecuado, porque aprender a programar ahora probablemente sea más fácil que nunca. Con asistentes virtuales como ChatGPT, no tienes que aprender en soledad. A lo largo de este libro, sugeriré formas de usar estas herramientas para acelerar tu aprendizaje.

Este libro es principalmente para personas que nunca han programado y para personas que tienen algo de experiencia en otro lenguaje de programación. Si tienes bastante experiencia en Python, quizá los primeros capítulos te parezcan demasiado lentos.

Uno de los retos de aprender a programar es que tienes que aprender *dos* lenguajes: uno es el propio lenguaje de programación; el otro es el vocabulario que usamos para hablar de programas. Si solo aprendes el lenguaje de programación, es probable que tengas problemas cuando necesites interpretar un mensaje de error, leer documentación, hablar con otra persona o usar asistentes virtuales. Si has programado algo, pero no has aprendido también este segundo lenguaje, espero que este libro te resulte útil.

Objetivos del libro

Al escribir este libro, intenté tener cuidado con el vocabulario. Defino cada término cuando aparece por primera vez. Y al final de cada capítulo hay un glosario que repasa los términos que se han introducido.

También intenté ser conciso. Cuanto menos esfuerzo mental requiera leer el libro, más capacidad tendrás para programar.

Pero no puedes aprender a programar solo leyendo un libro: tienes que practicar. Por eso, este libro incluye ejercicios al final de cada capítulo en los que puedes practicar lo que has aprendido.

Si lees con atención y trabajas en los ejercicios de forma constante, avanzarás. Pero te aviso desde ahora: aprender a programar no es fácil, e incluso para programadores con experiencia puede ser frustrante. A medida que avancemos, sugeriré estrategias para ayudarte a escribir programas correctos y arreglar los incorrectos.

Cómo navegar por el libro

Cada capítulo de este libro se basa en los anteriores, así que deberías leerlos en orden y dedicar tiempo a trabajar en los ejercicios antes de seguir adelante.

Los primeros seis capítulos presentan elementos básicos como la aritmética, los condicionales y los bucles. También presentan el concepto más importante de la programación, las funciones, y una forma potente de usarlas, la recursión.

Los capítulos 7 y 8 presentan strings, que pueden representar letras, palabras y frases, y algoritmos para trabajar con ellas.

Los capítulos 9 a 12 presentan las estructuras de datos principales de Python: listas, diccionarios y tuplas, que son herramientas potentes para escribir programas eficientes. El capítulo 12 presenta algoritmos para analizar texto y generar texto nuevo de forma aleatoria. Algoritmos como estos están en el núcleo de los modelos de lenguaje grandes (LLMs), así que este capítulo te dará una idea de cómo funcionan herramientas como ChatGPT.

El capítulo 13 trata sobre formas de almacenar datos a largo plazo: archivos y bases de datos. Como ejercicio, puedes escribir un programa que busque en un sistema de archivos y encuentre archivos duplicados.

Los capítulos 14 a 17 presentan la programación orientada a objetos (OOP), que es una forma de organizar programas y los datos con los que trabajan. Muchas librerías de Python están escritas con estilo orientado a objetos, así que estos capítulos te ayudarán a entender su diseño y a definir tus propios objetos.

El objetivo de este libro no es cubrir todo el lenguaje Python. En cambio, me centro en un subconjunto del lenguaje que ofrece la mayor capacidad con el menor número de conceptos. Aun así, Python tiene muchas características que puedes usar para resolver problemas comunes de forma eficiente. El capítulo 18 presenta algunas de estas características.

Por último, el capítulo 19 presenta mis reflexiones finales y sugerencias para continuar tu camino en la programación.

¿Qué hay de nuevo en la tercera edición?

Los cambios más grandes de esta edición fueron impulsados por dos tecnologías nuevas: los Jupyter notebooks y los asistentes virtuales.

Cada capítulo de este libro es un Jupyter notebook, que es un documento que contiene tanto texto normal como código. Para mí, eso facilita escribir el código, probarlo y mantenerlo coherente con el texto. Para ti, significa que puedes ejecutar el código, modificarlo y trabajar en los ejercicios, todo en un solo lugar. Las instrucciones para trabajar con los notebooks están en el primer capítulo.

El otro gran cambio es que he añadido consejos para trabajar con asistentes virtuales como ChatGPT y usarlos para acelerar tu aprendizaje. Cuando se publicó la edición anterior de este libro en 2016, los predecesores de estas herramientas eran mucho menos útiles y la mayoría de la gente no los conocía. Ahora son una herramienta estándar para la ingeniería de software, y creo que serán una herramienta transformadora para aprender a programar, y para aprender muchas otras cosas también.

Los demás cambios del libro fueron motivados por mis arrepentimientos sobre la segunda edición.

El primero es que no puse suficiente énfasis en las pruebas de software. Eso ya era una omisión lamentable en 2016, pero con la llegada de los asistentes virtuales, las pruebas automatizadas se han vuelto aún más importantes. Así que esta edición presenta las herramientas de testing más usadas de Python, `doctest` y `unittest`, e incluye varios ejercicios donde puedes practicar con ellas.

Mi otro arrepentimiento es que los ejercicios de la segunda edición eran desiguales: algunos eran más interesantes que otros y algunos eran demasiado difíciles. Pasar a Jupyter notebooks me ayudó a desarrollar y probar una secuencia de ejercicios más atractiva y eficaz.

En esta revisión, la secuencia de temas es casi la misma, pero reorganicé algunos capítulos y comprimí dos capítulos cortos en uno. Además, amplié la cobertura de strings para incluir expresiones regulares.

Algunos capítulos usan turtle graphics. En ediciones anteriores usé el módulo `turtle` de Python, pero por desgracia no funciona en Jupyter notebooks. Así que lo reemplacé por un nuevo turtle módulo que debería ser más fácil de usar.

Por último, reescribí una parte considerable del texto, aclarando lugares que lo necesitaban y recortando en lugares donde no fui tan conciso como podía ser.

Estoy muy orgulloso de esta nueva edición. ¡Espero que te guste!

Primeros pasos

Para la mayoría de los lenguajes de programación, incluido Python, hay muchas herramientas que puedes usar para escribir y ejecutar programas. Estas herramientas se llaman entornos de desarrollo integrados (IDEs). En general, hay dos tipos de IDEs:

- Algunos trabajan con archivos que contienen código, así que proporcionan herramientas para editar y ejecutar esos archivos.
- Otros trabajan principalmente con notebooks, que son documentos que contienen texto y código.

Para principiantes, recomiendo empezar con un entorno de desarrollo de notebooks como Jupyter.

Los notebooks de este libro están disponibles en un repositorio online en https://allendowney.github.io/ThinkPython.

Hay dos formas de usarlos:

- Puedes descargar los notebooks y ejecutarlos en tu propio ordenador. En ese caso, tienes que instalar Python y Jupyter, lo cual no es difícil, pero si quieres aprender Python puede ser frustrante pasar mucho tiempo instalando software.
- Una alternativa es ejecutar los notebooks en Colab, que es un entorno Jupyter que se ejecuta en un navegador web, así que no tienes que instalar nada. Colab está operado por Google y es gratuito.

Si estás empezando, te recomiendo encarecidamente que empieces con Colab.

Recursos para docentes

Si enseñas con este libro, aquí tienes algunos recursos que pueden resultarte útiles.

- Puedes encontrar notebooks con soluciones a los ejercicios en https://allendowney.github.io/ThinkPython, junto con enlaces a los recursos adicionales de abajo.
- Hay cuestionarios para cada capítulo, y un cuestionario final para todo el libro, disponibles bajo petición.
- *Teaching and Learning with Jupyter* es un libro online con sugerencias para usar Jupyter de forma eficaz en el aula. Puedes leer el libro en https://jupyter4edu.github.io/jupyter-edu-book
- Una de las mejores formas de usar notebooks es programación en vivo (live coding), donde un instructor escribe código y los estudiantes le siguen en sus propios notebooks. Para aprender sobre programación en vivo, y obtener otros grandes consejos sobre enseñanza de programación, recomiendo la formación para instructores ofrecida por The Carpentries, en https://carpentries.github.io/instructor-training

Agradecimientos

Muchas gracias a Jeff Elkner, que tradujo mi libro de Java a Python, lo que puso en marcha este proyecto y me presentó lo que ha acabado siendo mi lenguaje favorito. Gracias también a Chris Meyers, que contribuyó con varias secciones a *How to Think Like a Computer Scientist*.

Gracias a la Free Software Foundation por desarrollar la GNU Free Documentation License, que ayudó a hacer posible mi colaboración con Jeff y Chris, y gracias a Creative Commons por la licencia que uso ahora.

Gracias a quienes desarrollan y mantienen el lenguaje Python y las librerías que usé, incluido el módulo de Turtle graphics; las herramientas que usé para desarrollar el libro, incluidos Jupyter y JupyterBook; y los servicios que usé, incluidos ChatGPT, Copilot, Colab y GitHub.

Gracias a los editores de Lulu que trabajaron en *How to Think Like a Computer Scientist* y a los editores de O'Reilly Media que trabajaron en *Think Python*.

Un agradecimiento especial a los revisores técnicos de la segunda edición, Melissa Lewis y Luciano Ramalho, y de la tercera edición, Sam Lau y Luciano Ramalho (¡otra vez!). También agradezco a

Luciano haber desarrollado el turtle graphics módulo que uso en varios capítulos, llamado `jupyturtle`.

Gracias a todos los estudiantes que trabajaron con versiones anteriores de este libro y a todos los colaboradores que enviaron correcciones y sugerencias. Más de 100 lectores atentos y reflexivos han enviado sugerencias y correcciones durante los últimos años. Sus contribuciones, y su entusiasmo por este proyecto, han sido de enorme ayuda.

Si tienes una sugerencia o corrección, envía un email a `feedback@thinkpython.com`. Si incluyes al menos una parte de la frase en la que aparece el error, eso me facilita la búsqueda. Los números de página y sección también sirven, pero no son tan fáciles de manejar. ¡Gracias!

Think Python: 3.ª edición

Traducción al español por midudev (Miguel Ángel Durán).

Puedes pedir las versiones impresa y ebook de *Think Python 3e* en Bookshop.org y Amazon.

1. Bienvenida

Este es el Jupyter notebook del capítulo 1 de *Think Python*, 3.ª edición, de Allen B. Downey.

Si no conoces los Jupyter notebooks, haz clic aquí para ver una breve introducción.

Luego, si aún no estás ejecutando este notebook en Colab, haz clic aquí para ejecutar este notebook en Colab.

La siguiente celda descarga un archivo y ejecuta algo de código que se usa específicamente para este libro. Todavía no tienes que entender este código, pero deberías ejecutarlo antes de hacer cualquier otra cosa en este notebook. Recuerda que puedes ejecutar el código seleccionando la celda y pulsando el botón de reproducción (un triángulo dentro de un círculo) o manteniendo pulsada la tecla `Shift` y presionando `Enter`.

1. Programar como una forma de pensar

El primer objetivo de este libro es enseñarte a programar en Python. Pero aprender a programar implica aprender una nueva forma de pensar, así que el segundo objetivo de este libro es ayudarte a pensar como un científico de la computación. Esta forma de pensar combina algunas de las mejores características de las matemáticas, la ingeniería y las ciencias naturales. Como los matemáticos, los científicos de la computación usan lenguajes formales para expresar ideas – específicamente, cómputos. Como los ingenieros, diseñan cosas, ensamblan componentes en sistemas y evalúan las ventajas y desventajas de distintas alternativas. Como los científicos, observan el comportamiento de sistemas complejos, formulan hipótesis y prueban predicciones.

Empezaremos con los elementos más básicos de la programación y avanzaremos poco a poco. En este capítulo veremos cómo Python representa números, letras y palabras. Y aprenderás a realizar operaciones aritméticas.

También empezarás a aprender el vocabulario de la programación, incluyendo términos como operador, expresión, valor y tipo. Este vocabulario es importante – lo necesitarás para entender el resto del libro, comunicarte con otros programadores y usar y entender asistentes virtuales.

1.1. Operadores aritméticos

Un **operador aritmético** es un símbolo que representa un cálculo aritmético. Por ejemplo, el signo más, `+`, realiza una suma.

```
30 + 12
```

El signo menos, `-`, es el operador que realiza una resta.

```
43 - 1
```

El asterisco, `*`, realiza una multiplicación.

```
6 * 7
```

Y la barra inclinada, `/`, realiza una división:

```
84 / 2
```

Observa que el resultado de la división es `42.0` en lugar de `42`. Eso se debe a que hay dos tipos de números en Python:

- **enteros**, que representan números sin parte fraccionaria o decimal, y
- **números de coma flotante**, que representan enteros y números con punto decimal.

Si sumas, restas o multiplicas dos enteros, el resultado es un entero. Pero si divides dos enteros, el resultado es un número de coma flotante. Python proporciona otro operador, `//`, que realiza **división entera**. El resultado de la división entera siempre es un entero.

```
84 // 2
```

La división entera también se llama "división de piso" porque siempre redondea hacia abajo (hacia el "piso").

```
85 // 2
```

Por último, el operador `**` realiza exponenciación; es decir, eleva un número a una potencia:

```
7 ** 2
```

En algunos otros lenguajes, el acento circunflejo, `^`, se usa para la exponenciación, pero en Python es un operador bit a bit llamado XOR. Si no conoces los operadores bit a bit, el resultado puede ser inesperado:

```
7 ^ 2
```

No cubriré los operadores bit a bit en este libro, pero puedes leer sobre ellos en http://wiki.python.org/moin/BitwiseOperators.

1.2. Expresiones

Un conjunto de operadores y números se llama una **expresión**. Una expresión puede contener cualquier cantidad de operadores y números. Por ejemplo, aquí hay una expresión que contiene dos operadores.

```
6 + 6 ** 2
```

Observa que la exponenciación ocurre antes que la suma. Python sigue el orden de operaciones que quizá aprendiste en clase de matemáticas: la exponenciación ocurre antes que la multiplicación y la división, que ocurren antes que la suma y la resta.

En el siguiente ejemplo, la multiplicación ocurre antes que la suma.

```
12 + 5 * 6
```

Si quieres que la suma ocurra primero, puedes usar paréntesis.

```
(12 + 5) * 6
```

Toda expresión tiene un **valor**. Por ejemplo, la expresión `6 * 7` tiene el valor `42`.

1.3. Funciones aritméticas

Además de los operadores aritméticos, Python proporciona algunas **funciones** que trabajan con números. Por ejemplo, la función `round` toma un número de coma flotante y lo redondea al entero más cercano.

```
round(42.4)
```

```
round(42.6)
```

La función `abs` calcula el valor absoluto de un número. Para un número positivo, el valor absoluto es el propio número.

```
abs(42)
```

Para un número negativo, el valor absoluto es positivo.

```
abs(-42)
```

Cuando usamos una función como esta, decimos que estamos **llamando** a la función. Una expresión que llama a una función es una **llamada a función**.

Cuando llamas a una función, los paréntesis son obligatorios. Si los omites, obtienes un mensaje de error.

NOTA: La siguiente celda usa `%%expect`, que es un "magic command" de Jupyter que significa que esperamos que el código de esta celda produzca un error. Para más información sobre este tema, consulta la introducción a Jupyter notebook.

```
abs 42
```

Puedes ignorar la primera línea de este mensaje; no contiene ninguna información que necesitemos entender ahora mismo. La segunda línea es el código que contiene el error, con un acento circunflejo (`^`) debajo para indicar dónde se descubrió el error.

La última línea indica que se trata de un **error de sintaxis**, lo que significa que hay algo incorrecto en la estructura de la expresión. En este ejemplo, el problema es que una llamada a función requiere paréntesis.

Veamos qué ocurre si omites los paréntesis *y* el valor.

```
abs
```

Un nombre de función por sí solo es una expresión válida que tiene un valor. Cuando se muestra, el valor indica que `abs` es una función e incluye información adicional que explicaré más adelante.

1.4. Strings

Además de números, Python también puede representar secuencias de letras, que se llaman **strings** porque las letras están ensartadas como cuentas en un collar. Para escribir un string, podemos poner una secuencia de letras entre comillas rectas.

```
'Hello'
```

También es válido usar comillas dobles.

```
"world"
```

Las comillas dobles facilitan escribir un string que contiene un apóstrofo, que es el mismo símbolo que una comilla recta.

```
"it's a small "
```

Los strings también pueden contener espacios, signos de puntuación y dígitos.

```
'Well, '
```

El operador `+` funciona con strings; une dos strings en un solo string, lo que se llama **concatenación**

```
'Well, ' + "it's a small " + 'world.'
```

El operador `*` también funciona con strings; hace varias copias de un string y las concatena.

```
'Spam, ' * 4
```

Los otros operadores aritméticos no funcionan con strings.

Python proporciona una función llamada `len` que calcula la longitud de un string.

```
len('Spam')
```

Observa que `len` cuenta las letras entre las comillas, pero no las comillas.

Cuando crees un string, asegúrate de usar comillas rectas. La comilla invertida, también conocida como backtick, causa un error de sintaxis.

```
`Hello`
```

Las comillas tipográficas, también conocidas como comillas curvas, tampoco son válidas.

```
‘Hello’
```

1.5. Valores y tipos

Hasta ahora hemos visto tres clases de valores:

- `2` es un entero,
- `42.0` es un número de coma flotante, y
- `'Hello'` es un string.

Una clase de valor se llama **tipo**. Todo valor tiene un tipo – o a veces decimos que “pertenece a” un tipo.

Python proporciona una función llamada `type` que te dice el tipo de cualquier valor. El tipo de un entero es `int`.

```
type(2)
```

El tipo de un número de coma flotante es `float`.

```
type(42.0)
```

Y el tipo de un string es `str`.

```
type('Hello, World!')
```

Los tipos `int`, `float` y `str` se pueden usar como funciones. Por ejemplo, `int` puede tomar un número de coma flotante y convertirlo en un entero (siempre redondeando hacia abajo).

```
int(42.9)
```

Y `float` puede convertir un entero en un valor de punto flotante.

```
float(42)
```

Ahora, aquí hay algo que puede ser confuso. ¿Qué obtienes si pones una secuencia de dígitos entre comillas?

```
'126'
```

Parece un número, pero en realidad es un string.

```
type('126')
```

Si intentas usarlo como un número, puede que obtengas un error.

```
'126' / 3
```

Este ejemplo genera un `TypeError`, lo que significa que los valores de la expresión, que se llaman **operandos**, tienen el tipo incorrecto. El mensaje de error indica que el operador `/` no admite los tipos de estos valores, que son `str` e `int`.

Si tienes un string que contiene dígitos, puedes usar `int` para convertirlo en un entero.

```
int('126') / 3
```

Si tienes un string que contiene dígitos y un punto decimal, puedes usar `float` para convertirlo en un número de coma flotante.

```
float('12.6')
```

Cuando escribes un entero grande, puede que te sientas tentado a usar comas entre grupos de dígitos, como en `1,000,000`. Esta es una expresión válida en Python, pero el resultado no es un entero.

```
1,000,000
```

Python interpreta `1,000,000` como una secuencia de enteros separados por comas. Aprenderemos más sobre este tipo de secuencia más adelante.

Puedes usar guiones bajos para que los números grandes sean más fáciles de leer.

```
1_000_000
```

1.6. Lenguajes formales y naturales

Los **lenguajes naturales** son los idiomas que hablan las personas, como inglés, español y francés. No fueron diseñados por personas; evolucionaron de forma natural.

Los **lenguajes formales** son lenguajes diseñados por personas para aplicaciones específicas. Por ejemplo, la notación que usan los matemáticos es un lenguaje formal especialmente bueno para representar relaciones entre números y símbolos. De forma similar, los lenguajes de programación son lenguajes formales diseñados para expresar cómputos.

Aunque los lenguajes formales y naturales tienen algunas características en común, hay diferencias importantes:

- Ambigüedad: Los lenguajes naturales están llenos de ambigüedad, que las personas manejan usando pistas contextuales y otra información. Los lenguajes formales están diseñados para ser casi o completamente no ambiguos, lo que significa que cualquier programa tiene exactamente un significado, independientemente del contexto.
- Redundancia: Para compensar la ambigüedad y reducir malentendidos, los lenguajes naturales usan redundancia. Como resultado, a menudo son verbosos. Los lenguajes formales son menos redundantes y más concisos.

- Literalidad: Los lenguajes naturales están llenos de modismos y metáforas. Los lenguajes formales significan exactamente lo que dicen.

Como todos crecemos hablando lenguajes naturales, a veces cuesta adaptarse a los lenguajes formales. Los lenguajes formales son más densos que los lenguajes naturales, así que lleva más tiempo leerlos. Además, la estructura es importante, por lo que no siempre es mejor leer de arriba abajo, de izquierda a derecha. Por último, los detalles importan. Pequeños errores de ortografía y puntuación, que puedes tolerar en los lenguajes naturales, pueden marcar una gran diferencia en un lenguaje formal.

1.7. Debugging

Los programadores cometen errores. Por razones caprichosas, los errores de programación se llaman **bugs** y el proceso de encontrarlos se llama **debugging**.

Programar, y especialmente debugging, a veces provoca emociones intensas. Si estás luchando con un bug difícil, puede que sientas enfado, tristeza o vergüenza.

Prepararte para estas reacciones puede ayudarte a manejarlas. Un enfoque es pensar en la computadora como un empleado con ciertas fortalezas, como velocidad y precisión, y debilidades particulares, como falta de empatía e incapacidad para captar el panorama general.

Tu trabajo es ser un buen gerente: encontrar formas de aprovechar las fortalezas y mitigar las debilidades. Y encontrar formas de usar tus emociones para involucrarte con el problema, sin dejar que tus reacciones interfieran con tu capacidad de trabajar eficazmente.

Aprender a debug puede ser frustrante, pero es una habilidad valiosa que resulta útil para muchas actividades más allá de la programación. Al final de cada capítulo hay una sección, como esta, con mis sugerencias para debugging. ¡Espero que te ayuden!

1.8. Glosario

operador aritmético: Un símbolo, como `+` y `*`, que denota una operación aritmética como suma o multiplicación.

entero: Un tipo que representa números sin parte fraccionaria o decimal.

número de coma flotante: Un tipo que representa enteros y números con partes decimales.

división entera: Un operador, `//`, que divide dos números y redondea hacia abajo a un entero.

expresión: Una combinación de variables, valores y operadores.

valor: Un entero, número de coma flotante o string – u otro de los tipos de valores que veremos más adelante.

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

llamada a función: Una expresión – o parte de una expresión – que ejecuta una función. Consiste en el nombre de la función seguido de una lista de argumentos entre paréntesis.

error de sintaxis: Un error en un programa que hace imposible analizarlo – y por lo tanto imposible ejecutarlo.

string: Un tipo que representa secuencias de caracteres.

concatenación: Unir dos strings extremo con extremo.

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

operando: Uno de los valores sobre los que opera un operador.

lenguaje natural: Cualquiera de los idiomas que hablan las personas y que evolucionaron de forma natural.

lenguaje formal: Cualquiera de los lenguajes que las personas han diseñado para propósitos específicos, como representar ideas matemáticas o programas de computadora. Todos los lenguajes de programación son lenguajes formales.

bug: Un error en un programa.

debugging: El proceso de encontrar y corregir errores.

1.9. Ejercicios

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

1.9.1. Pregúntale a un asistente virtual

A medida que avances en este libro, hay varias formas en las que puedes usar un asistente virtual o chatbot para ayudarte a aprender.

- Si quieres aprender más sobre un tema del capítulo, o algo no está claro, puedes pedir una explicación.
- Si tienes dificultades con alguno de los ejercicios, puedes pedir ayuda.

En cada capítulo sugeriré ejercicios que puedes hacer con un asistente virtual, pero te animo a probar cosas por tu cuenta y ver qué te funciona.

Aquí tienes algunos temas sobre los que podrías preguntar a un asistente virtual:

- Antes mencioné los operadores bit a bit, pero no expliqué por qué el valor de `7 ^ 2` es 5. Prueba a preguntar "¿Cuáles son los operadores bit a bit en Python?" o "¿Cuál es el valor de `7 XOR 2`?"
- También mencioné el orden de operaciones. Para más detalles, pregunta "¿Cuál es el orden de las operaciones en Python?"
- La función `round`, que usamos para redondear un número de coma flotante al entero más cercano, puede tomar un segundo argumento. Prueba a preguntar "¿Cuáles son los argumentos de la función round?" o "¿Cómo redondeo pi a tres decimales?"
- Hay un operador aritmético más que no mencioné; prueba a preguntar "¿Qué es el operador módulo en Python?"

La mayoría de los asistentes virtuales saben sobre Python, así que responden preguntas como estas de forma bastante fiable. Pero recuerda que estas herramientas cometen errores. Si obtienes código de un chatbot, ¡pruébalo!

1.9.2. Ejercicio

Quizá te preguntes qué hace `round` si un número termina en `0.5`. La respuesta es que a veces redondea hacia arriba y a veces hacia abajo. Prueba estos ejemplos y mira si puedes averiguar qué regla sigue.

```
round(42.5)
```

```
round(43.5)
```

Si tienes curiosidad, pregunta a un asistente virtual: "Si un número termina en 0.5, ¿Python redondea hacia arriba o hacia abajo?"

1.9.3. Ejercicio

Cuando aprendes una característica nueva, deberías probarla y cometer errores a propósito. Así aprendes los mensajes de error y, cuando los vuelvas a ver, sabrás qué significan. Es mejor cometer errores ahora y deliberadamente que más tarde y accidentalmente.

1. Puedes usar un signo menos para crear un número negativo como `-2`. ¿Qué ocurre si pones un signo más delante de un número? ¿Y con `2++2`?
2. ¿Qué ocurre si tienes dos valores sin operador entre ellos, como `4 2`?
3. Si llamas a una función como `round(42.5)`, ¿qué ocurre si omites uno o ambos paréntesis?

1.9.4. Ejercicio

Recuerda que toda expresión tiene un valor, todo valor tiene un tipo, y podemos usar la función `type` para encontrar el tipo de cualquier valor.

¿Cuál es el tipo del valor de las siguientes expresiones? Haz tu mejor conjetura para cada una y luego usa `type` para averiguarlo.

- `765`
- `2.718`
- `'2 pi'`

- `abs(-7)`
- `abs(-7.0)`
- `abs`
- `int`
- `type`

1.9.5. Ejercicio

Las siguientes preguntas te dan la oportunidad de practicar escribiendo expresiones aritméticas.

1. ¿Cuántos segundos hay en 42 minutos y 42 segundos?
2. ¿Cuántas millas hay en 10 kilómetros? Pista: hay 1,61 kilómetros en una milla.
3. Si corres una carrera de 10 kilómetros en 42 minutos y 42 segundos, ¿cuál es tu ritmo medio en segundos por milla?
4. ¿Cuál es tu ritmo medio en minutos y segundos por milla?
5. ¿Cuál es tu velocidad media en millas por hora?

Si ya sabes sobre variables, puedes usarlas para este ejercicio. Si no, puedes hacer el ejercicio sin ellas – y luego las veremos en el siguiente capítulo.

2. Variables y sentencias

En el capítulo anterior usamos operadores para escribir expresiones que realizan cálculos aritméticos.

En este capítulo aprenderás sobre variables y sentencias, la sentencia `import` y la función `print`. Y presentaré más vocabulario que usamos para hablar de programas, incluyendo “argumento” y “módulo”.

2.1. Variables

Una **variable** es un nombre que se refiere a un valor. Para crear una variable, podemos escribir una **sentencia de asignación** como esta.

```
n = 17
```

Una sentencia de asignación tiene tres partes: el nombre de la variable a la izquierda, el operador igual, `=`, y una expresión a la derecha. En este ejemplo, la expresión es un entero. En el siguiente ejemplo, la expresión es un número de punto flotante.

```
pi = 3.141592653589793
```

Y en el siguiente ejemplo, la expresión es un string.

```
message = 'And now for something completely different'
```

Cuando ejecutas una sentencia de asignación, no hay salida. Python crea la variable y le da un valor, pero la sentencia de asignación no tiene ningún efecto visible. Sin embargo, después de crear una variable, puedes usarla como una expresión. Así que podemos mostrar el valor de `message` así:

```
message
```

También puedes usar una variable como parte de una expresión con operadores aritméticos.

```
n + 25
```

```
2 * pi
```

Y puedes usar una variable cuando llamas a una función.

```
round(pi)
```

```
len(message)
```

2.2. Diagramas de estado

Una forma común de representar variables en papel es escribir el nombre con una flecha apuntando a su valor.

```
import math

from diagram import make_binding, Frame

binding = make_binding("message", 'And now for something completely different')
binding2 = make_binding("n", 17)
binding3 = make_binding("pi", 3.141592653589793)

frame = Frame([binding2, binding3, binding])
```

```
from diagram import diagram, adjust


width, height, x, y = [3.62, 1.01, 0.6, 0.76]
ax = diagram(width, height)
bbox = frame.draw(ax, x, y, dy=-0.25)
# adjust(x, y, bbox)
```

Este tipo de figura se llama **diagrama de estado** porque muestra en qué estado está cada una de las variables (piensa en ello como el estado mental de la variable). Usaremos diagramas de estado a lo largo del libro para representar un modelo de cómo Python almacena variables y sus valores.

2.3. Nombres de variables

Los nombres de variables pueden ser tan largos como quieras. Pueden contener letras y números, pero no pueden comenzar con un número. Es válido usar letras mayúsculas, pero lo convencional es usar solo minúsculas para los nombres de variables.

El único signo de puntuación que puede aparecer en un nombre de variable es el carácter de guion bajo, `_`. Se usa a menudo en nombres con varias palabras, como `your_name` o `airspeed_of_unladen_swallow`.

Si le das a una variable un nombre no válido, obtienes un error de sintaxis. El nombre `million!` no es válido porque contiene puntuación.

```
million! = 1000000
```

`76trombones` no es válido porque empieza con un número.

```
76trombones = 'big parade'
```

`class` tampoco es válido, pero puede que no sea obvio por qué.

```
class = 'Self-Defence Against Fresh Fruit'
```

Resulta que `class` es una **palabra clave**, que es una palabra especial usada para especificar la estructura de un programa. Las palabras clave no se pueden usar como nombres de variables.

Aquí tienes una lista completa de las palabras clave de Python:

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

```
from keyword import kwlist

len(kwlist)
```

No tienes que memorizar esta lista. En la mayoría de los entornos de desarrollo, las palabras clave se muestran en un color diferente; si intentas usar una como nombre de variable, lo sabrás.

2.4. La sentencia import

Para usar algunas características de Python, tienes que **importarlas**. Por ejemplo, la siguiente sentencia importa el módulo `math`.

```
import math
```

Un **módulo** es una colección de variables y funciones. El módulo math proporciona una variable llamada `pi` que contiene el valor de la constante matemática denotada π. Podemos mostrar su valor así.

```
math.pi
```

Para usar una variable de un módulo, tienes que usar el **operador punto** (`.`) entre el nombre del módulo y el nombre de la variable.

El módulo math también contiene funciones. Por ejemplo, `sqrt` calcula raíces cuadradas.

```
math.sqrt(25)
```

Y `pow` eleva un número a la potencia de un segundo número.

```
math.pow(5, 2)
```

En este punto hemos visto dos formas de elevar un número a una potencia: podemos usar la función `math.pow` o el operador de exponenciación, `**`. Cualquiera de las dos está bien, pero el operador se usa más a menudo que la función.

2.5. Expresiones y sentencias

Hasta ahora hemos visto algunos tipos de expresiones. Una expresión puede ser un solo valor, como un entero, un número de punto flotante o un string. También puede ser una colección de valores y operadores. Y puede incluir nombres de variables y llamadas a funciones. Aquí hay una expresión que incluye varios de estos elementos.

```
19 + n + round(math.pi) * 2
```

También hemos visto algunos tipos de sentencias. Una **sentencia** es una unidad de código que tiene un efecto, pero no un valor. Por ejemplo, una sentencia de asignación crea una variable y le da un valor, pero la sentencia en sí no tiene valor.

```
n = 17
```

De forma similar, una sentencia import tiene un efecto – importa un módulo para que podamos usar las variables y funciones que contiene – pero no tiene efecto visible.

```
import math
```

Calcular el valor de una expresión se llama **evaluación**. Ejecutar una sentencia se llama **ejecución**.

2.6. La función print

Cuando evalúas una expresión, se muestra el resultado.

```
n + 1
```

Pero si evalúas más de una expresión, solo se muestra el valor de la última.

```
n + 2
n + 3
```

Para mostrar más de un valor, puedes usar la función `print`.

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

También funciona con números de punto flotante y strings.

```
print('The value of pi is approximately')
print(math.pi)
```

También puedes usar una secuencia de expresiones separadas por comas.

```
print('The value of pi is approximately', math.pi)
```

Observa que la función `print` coloca un espacio entre los valores.

2.7. Argumentos

Cuando llamas a una función, la expresión entre paréntesis se llama un **argumento**. Normalmente explicaría por qué, pero en este caso el significado técnico de un término no tiene casi nada que ver con el significado común de la palabra, así que ni siquiera lo intentaré.

Algunas de las funciones que hemos visto hasta ahora toman solo un argumento, como `int`.

```
int('101')
```

Algunas toman dos, como `math.pow`.

```
math.pow(5, 2)
```

Algunas pueden tomar argumentos adicionales que son opcionales. Por ejemplo, `int` puede tomar un segundo argumento que especifica la base del número.

```
int('101', 2)
```

La secuencia de dígitos `101` en base 2 representa el número 5 en base 10.

`round` también toma un segundo argumento opcional, que es el número de posiciones decimales al que redondear.

```
round(math.pi, 3)
```

Algunas funciones pueden tomar cualquier número de argumentos, como `print`.

```
print('Any', 'number', 'of', 'arguments')
```

Si llamas a una función y proporcionas demasiados argumentos, eso es un `TypeError`.

```
float('123.0', 2)
```

Si proporcionas muy pocos argumentos, eso también es un `TypeError`.

```
math.pow(2)
```

Y si proporcionas un argumento con un tipo que la función no puede manejar, eso también es un `TypeError`.

```
math.sqrt('123')
```

Este tipo de comprobación puede ser molesto cuando estás empezando, pero te ayuda a detectar y corregir errores.

2.8. Comentarios

A medida que los programas se hacen más grandes y complicados, se vuelven más difíciles de leer. Los lenguajes formales son densos, y a menudo es difícil mirar un fragmento de código y averiguar qué hace y por qué.

Por esta razón, es buena idea añadir notas a tus programas para explicar en lenguaje natural qué está haciendo el programa. Estas notas se llaman **comentarios**, y empiezan con el símbolo `#`.

```
# number of seconds in 42:42
seconds = 42 * 60 + 42
```

En este caso, el comentario aparece en una línea por sí solo. También puedes poner comentarios al final de una línea:

```
miles = 10 / 1.61     # 10 kilometers in miles
```

Todo desde el `#` hasta el final de la línea se ignora—no tiene efecto en la ejecución del programa.

Los comentarios son más útiles cuando documentan características no obvias del código. Es razonable suponer que el lector puede averiguar *qué* hace el código; es más útil explicar *por qué*.

Este comentario es redundante con el código y no sirve:

```
v = 8     # assign 8 to v
```

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

```
v = 8     # velocity in miles per hour
```

Los buenos nombres de variables pueden reducir la necesidad de comentarios, pero los nombres largos pueden hacer que las expresiones complejas sean difíciles de leer, así que hay una compensación.

2.9. Debugging

En un programa pueden ocurrir tres tipos de errores: errores de sintaxis, errores en tiempo de ejecución y errores semánticos. Es útil distinguirlos para localizarlos más rápidamente.

- **Error de sintaxis**: "Sintaxis" se refiere a la estructura de un programa y a las reglas sobre esa estructura. Si hay un error de sintaxis en cualquier parte de tu programa, Python no ejecuta el programa. Muestra un mensaje de error inmediatamente.
- **Error en tiempo de ejecución**: Si no hay errores de sintaxis en tu programa, puede empezar a ejecutarse. Pero si algo sale mal, Python muestra un mensaje de error y se detiene. Este tipo de error se llama error en tiempo de ejecución. También se llama una **excepción** porque indica que ha ocurrido algo excepcional.
- **Error semántico**: El tercer tipo de error es "semántico", lo que significa relacionado con el significado. Si hay un error semántico en tu programa, se ejecuta sin generar mensajes de error, pero no hace lo que pretendías. Identificar errores semánticos puede ser complicado

porque requiere que trabajes hacia atrás mirando la salida del programa e intentando averiguar qué está haciendo.

Como hemos visto, un nombre de variable no válido es un error de sintaxis.

```
million! = 1000000
```

Si usas un operador con un tipo que no admite, eso es un error en tiempo de ejecución.

```
'126' / 3
```

Por último, aquí tienes un ejemplo de error semántico. Supongamos que queremos calcular la media de `1` y `3`, pero olvidamos el orden de operaciones y escribimos esto:

```
1 + 3 / 2
```

Cuando se evalúa esta expresión, no produce un mensaje de error, así que no hay error de sintaxis ni error en tiempo de ejecución. Pero el resultado no es la media de `1` y `3`, así que el programa no es correcto. Este es un error semántico porque el programa se ejecuta pero no hace lo que se pretendía.

2.10. Glosario

variable: Un nombre que se refiere a un valor.

sentencia de asignación: Una sentencia que asigna un valor a una variable.

diagrama de estado: Una representación gráfica de un conjunto de variables y los valores a los que se refieren.

palabra clave: Una palabra especial usada para especificar la estructura de un programa.

sentencia de importación: Una sentencia que lee un archivo de módulo para que podamos usar las variables y funciones que contiene.

módulo: Un archivo que contiene código Python, incluyendo definiciones de funciones y a veces otras sentencias.

operador punto: El operador, `.`, usado para acceder a una función en otro módulo especificando el nombre del módulo seguido de un punto y el nombre de la función.

evaluar: Realizar las operaciones de una expresión para calcular un valor.

sentencia: Una o más líneas de código que representan una orden o acción.

ejecutar: Ejecutar una sentencia y hacer lo que dice.

argumento: Un valor proporcionado a una función cuando se llama a la función.

comentario: Texto incluido en un programa que proporciona información sobre el programa pero no tiene efecto en su ejecución.

error en tiempo de ejecución: Un error que hace que un programa muestre un mensaje de error y termine.

excepción: Un error que se detecta mientras el programa se está ejecutando.

error semántico: Un error que hace que un programa haga algo incorrecto, pero no muestre un mensaje de error.

2.11. Ejercicios

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

2.11.1. Pregúntale a un asistente virtual

De nuevo, te animo a usar un asistente virtual para aprender más sobre cualquiera de los temas de este capítulo.

Si tienes curiosidad por alguna de las palabras clave que enumeré, podrías preguntar "¿Por qué class es una palabra clave?" o "¿Por qué los nombres de variables no pueden ser palabras clave?"

Quizá hayas notado que `int`, `float` y `str` no son palabras clave de Python. Son variables que representan tipos, y se pueden usar como funciones. Así que es *legal* tener una variable o función con uno de esos nombres, pero se desaconseja firmemente. Pregunta a un asistente: "¿Por qué es mala idea usar int, float y str como nombres de variables?"

Pregunta también: "¿Cuáles son las funciones incorporadas de Python?" Si tienes curiosidad por alguna de ellas, pide más información.

En este capítulo importamos el módulo `math` y usamos algunas de las variables y funciones que proporciona. Pregunta a un asistente: "¿Qué variables y funciones hay en el módulo math?" y "Además de math, ¿qué módulos se consideran parte del núcleo de Python?"

2.11.2. Ejercicio

Repitiendo mi consejo del capítulo anterior: siempre que aprendas una característica nueva, deberías cometer errores a propósito para ver qué falla.

- Hemos visto que `n = 17` es válido. ¿Qué pasa con `17 = n`?
- ¿Y con `x = y = 1`?
- En algunos lenguajes, cada sentencia termina con punto y coma (`;`). ¿Qué ocurre si pones un punto y coma al final de una sentencia de Python?
- ¿Qué pasa si pones un punto al final de una sentencia?
- ¿Qué ocurre si escribes mal el nombre de un módulo e intentas importar `maath`?

2.11.3. Ejercicio

Practica usando el intérprete de Python como calculadora:

Parte 1. El volumen de una esfera con radio r es $\frac{4}{3}\pi r^3$. ¿Cuál es el volumen de una esfera con radio 5? Empieza con una variable llamada `radius` y luego asigna el resultado a una variable llamada `volume`. Muestra el resultado. Añade comentarios para indicar que `radius` está en centímetros y `volume` en centímetros cúbicos.

Parte 2. Una regla de trigonometría dice que para cualquier valor de x, $(\cos x)^2 + (\sin x)^2 = 1$. Veamos si es cierta para un valor específico de x, como 42.

Crea una variable llamada `x` con este valor. Luego usa `math.cos` y `math.sin` para calcular el seno y el coseno de x, y la suma de sus cuadrados.

El resultado debería estar cerca de 1. Puede que no sea exactamente 1 porque la aritmética de punto flotante no es exacta—solo es aproximadamente correcta.

Parte 3. Además de `pi`, la otra variable definida en el módulo `math` es `e`, que representa la base del logaritmo natural, escrita en notación matemática como e. Si no conoces este valor, pregunta a un asistente virtual "¿Qué es `math.e`?" Ahora calculemos e^2 de tres formas:

- Usa `math.e` y el operador de exponenciación (`**`).
- Usa `math.pow` para elevar `math.e` a la potencia `2`.
- Usa `math.exp`, que toma como argumento un valor, x, y calcula e^x.

Puede que notes que el último resultado es ligeramente diferente de los otros dos. Mira si puedes averiguar cuál es correcto.

3. Funciones

En el capítulo anterior usamos varias funciones proporcionadas por Python, como `int` y `float`, y algunas proporcionadas por el módulo `math`, como `sqrt` y `pow`. En este capítulo aprenderás a crear tus propias funciones y a ejecutarlas. Y veremos cómo una función puede llamar a otra. Como ejemplos, mostraremos letras de canciones de Monty Python. Estos ejemplos absurdos

demuestran una característica importante: la capacidad de escribir tus propias funciones es la base de la programación.

Este capítulo también introduce una nueva sentencia, el bucle `for`, que se usa para repetir un cálculo.

3.1. Definir nuevas funciones

Una **definición de función** especifica el nombre de una nueva función y la secuencia de sentencias que se ejecutan cuando se llama a la función. Aquí tienes un ejemplo:

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")
```

`def` es una palabra clave que indica que esto es una definición de función. El nombre de la función es `print_lyrics`. Cualquier cosa que sea un nombre de variable válido también es un nombre de función válido.

Los paréntesis vacíos después del nombre indican que esta función no recibe argumentos.

La primera línea de la definición de función se llama **encabezado**; el resto se llama **cuerpo**. El encabezado tiene que terminar con dos puntos y el cuerpo tiene que estar indentado. Por convención, la indentación es siempre de cuatro espacios. El cuerpo de esta función son dos sentencias `print`; en general, el cuerpo de una función puede contener cualquier número de sentencias de cualquier tipo.

Definir una función crea un **objeto función**, que podemos mostrar así.

```
print_lyrics
```

La salida indica que `print_lyrics` es una función que no recibe argumentos. `__main__` es el nombre del módulo que contiene `print_lyrics`.

Ahora que hemos definido una función, podemos llamarla del mismo modo que llamamos a las funciones incorporadas.

```
print_lyrics()
```

Cuando la función se ejecuta, ejecuta las sentencias del cuerpo, que muestran las dos primeras líneas de “The Lumberjack Song”.

3.2. Parámetros

Algunas de las funciones que hemos visto requieren argumentos; por ejemplo, cuando llamas a `abs` pasas un número como argumento. Algunas funciones reciben más de un argumento; por ejemplo, `math.pow` recibe dos: la base y el exponente.

Aquí tienes la definición de una función que recibe un argumento.

```
def print_twice(string):
    print(string)
    print(string)
```

El nombre de variable entre paréntesis es un **parámetro**. Cuando se llama a la función, el valor del argumento se asigna al parámetro. Por ejemplo, podemos llamar a `print_twice` así.

```
print_twice('Dennis Moore, ')
```

Ejecutar esta función tiene el mismo efecto que asignar el argumento al parámetro y luego ejecutar el cuerpo de la función, así.

```
string = 'Dennis Moore, '
print(string)
print(string)
```

También puedes usar una variable como argumento.

```
line = 'Dennis Moore, '
print_twice(line)
```

En este ejemplo, el valor de `line` se asigna al parámetro `string`.

3.3. Llamar funciones

Una vez que has definido una función, puedes usarla dentro de otra función. Para demostrarlo, escribiremos funciones que imprimen la letra de "The Spam Song" (https://www.songfacts.com/lyrics/monty-python/the-spam-song).

> Spam, Spam, Spam, Spam,
> Spam, Spam, Spam, Spam,
> Spam, Spam,
> (Lovely Spam, Wonderful Spam!)
> Spam, Spam,

Empezaremos con la siguiente función, que recibe dos parámetros.

```
def repeat(word, n):
    print(word * n)
```

Podemos usar esta función para imprimir la primera línea de la canción, así.

```
spam = 'Spam, '
repeat(spam, 4)
```

Para mostrar las dos primeras líneas, podemos definir una nueva función que use `repeat`.

```
def first_two_lines():
    repeat(spam, 4)
    repeat(spam, 4)
```

Y luego llamarla así.

```
first_two_lines()
```

Para mostrar las últimas tres líneas, podemos definir otra función, que también usa `repeat`.

```
def last_three_lines():
    repeat(spam, 2)
    print('(Lovely Spam, Wonderful Spam!)')
    repeat(spam, 2)
```

```
last_three_lines()
```

Por último, podemos unirlo todo con una función que imprime la estrofa completa.

```
def print_verse():
    first_two_lines()
    last_three_lines()
```

```
print_verse()
```

Cuando ejecutamos `print_verse`, llama a `first_two_lines`, que llama a `repeat`, que llama a `print`. Son muchas funciones.

Por supuesto, podríamos haber hecho lo mismo con menos funciones, pero el objetivo de este ejemplo es mostrar cómo las funciones pueden trabajar juntas.

3.4. Repetición

Si queremos mostrar más de una estrofa, podemos usar una sentencia `for`. Aquí tienes un ejemplo sencillo.

```
for i in range(2):
    print(i)
```

La primera línea es un encabezado que termina con dos puntos. La segunda línea es el cuerpo, que tiene que estar indentado.

El encabezado empieza con la palabra clave `for`, una nueva variable llamada `i` y otra palabra clave, `in`. Usa la función `range` para crear una secuencia de dos valores, que son `0` y `1`. En Python, cuando empezamos a contar, normalmente empezamos desde `0`.

Cuando la sentencia `for` se ejecuta, asigna el primer valor de `range` a `i` y luego ejecuta la función `print` en el cuerpo, que muestra `0`.

Cuando llega al final del cuerpo, vuelve al encabezado, por eso esta sentencia se llama **bucle**. La segunda vez que recorre el bucle, asigna el siguiente valor de `range` a `i` y lo muestra. Entonces, como ese es el último valor de `range`, el bucle termina.

Así es como podemos usar un bucle `for` para imprimir dos estrofas de la canción.

```
for i in range(2):
    print("Verse", i)
    print_verse()
    print()
```

Puedes poner un bucle `for` dentro de una función. Por ejemplo, `print_n_verses` recibe un parámetro llamado `n`, que tiene que ser un entero, y muestra el número dado de estrofas.

```
def print_n_verses(n):
    for i in range(n):
        print_verse()
        print()
```

En este ejemplo, no usamos `i` en el cuerpo del bucle, pero de todos modos tiene que haber un nombre de variable en el encabezado.

3.5. Las variables y los parámetros son locales

Cuando creas una variable dentro de una función, es **local**, lo que significa que solo existe dentro de la función. Por ejemplo, la siguiente función recibe dos argumentos, los concatena e imprime el resultado dos veces.

```
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

Aquí tienes un ejemplo que la usa:

```
line1 = 'Always look on the '
line2 = 'bright side of life.'
cat_twice(line1, line2)
```

Cuando `cat_twice` se ejecuta, crea una variable local llamada `cat`, que se destruye cuando termina la función. Si intentamos mostrarla, obtenemos un `NameError`:

```
print(cat)
```

Fuera de la función, `cat` no está definida.

Los parámetros también son locales. Por ejemplo, fuera de `cat_twice`, no existe nada llamado `part1` o `part2`.

3.6. Diagramas de pila

Para llevar la cuenta de qué variables se pueden usar en cada lugar, a veces es útil dibujar un **diagrama de pila**. Como los diagramas de estado, los diagramas de pila muestran el valor de cada variable, pero también muestran la función a la que pertenece cada variable.

Cada función se representa mediante un **marco**. Un marco es una caja con el nombre de una función por fuera y los parámetros y variables locales de la función por dentro.

Aquí tienes el diagrama de pila del ejemplo anterior.

```
from diagram import make_frame, Stack

d1 = dict(line1=line1, line2=line2)
frame1 = make_frame(d1, name='__main__', dy=-0.3, loc='left')

d2 = dict(part1=line1, part2=line2, cat=line1+line2)
frame2 = make_frame(d2, name='cat_twice', dy=-0.3,
                    offsetx=0.03, loc='left')

d3 = dict(string=line1+line2)
frame3 = make_frame(d3, name='print_twice',
                    offsetx=0.04, offsety=-0.3, loc='left')

d4 = {"?": line1+line2}
frame4 = make_frame(d4, name='print',
                    offsetx=-0.22, offsety=0, loc='left')

stack = Stack([frame1, frame2, frame3, frame4], dy=-0.8)
```

```
from diagram import diagram, adjust


width, height, x, y = [3.77, 2.9, 1.1, 2.65]
ax = diagram(width, height)
bbox = stack.draw(ax, x, y)
# adjust(x, y, bbox)

import matplotlib.pyplot as plt
plt.savefig('chap03_stack_diagram.png', dpi=300)
```

Los marcos se organizan en un stack que indica qué función llamó a cuál, y así sucesivamente. Leyendo desde abajo, `print` fue llamada por `print_twice`, que fue llamada por `cat_twice`, que fue llamada por `__main__`, que es un nombre especial para el marco superior. Cuando creas una variable fuera de cualquier función, pertenece a `__main__`.

En el marco de `print`, el signo de interrogación indica que no sabemos el nombre del parámetro. Si tienes curiosidad, pregunta a un asistente virtual: «¿Cuáles son los parámetros de la función `print` de Python?

3.7. Tracebacks

Cuando ocurre un error en tiempo de ejecución dentro de una función, Python muestra el nombre de la función que se estaba ejecutando, el nombre de la función que la llamó, y así sucesivamente,

subiendo por el stack. Para ver un ejemplo, definiré una versión de `print_twice` que contiene un error: intenta imprimir `cat`, que es una variable local de otra función.

```
def print_twice(string):
    print(cat)            # NameError
    print(cat)
```

Ahora esto es lo que ocurre cuando ejecutamos `cat_twice`.

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs, including a traceback.

%xmode Verbose
```

```
cat_twice(line1, line2)
```

El mensaje de error incluye un **traceback**, que muestra la función que se estaba ejecutando cuando ocurrió el error, la función que la llamó, y así sucesivamente. En este ejemplo, muestra que `cat_twice` llamó a `print_twice`, y que el error ocurrió en un `print_twice`.

El orden de las funciones en el traceback es el mismo que el orden de los marcos en el diagrama de pila. La función que se estaba ejecutando está abajo del todo.

3.8. ¿Por qué funciones?

Puede que todavía no esté claro por qué merece la pena dividir un programa en funciones. Hay varias razones:

- Crear una nueva función te da la oportunidad de nombrar un grupo de sentencias, lo que hace que tu programa sea más fácil de leer y depurar.
- Las funciones pueden hacer que un programa sea más pequeño al eliminar código repetitivo. Más adelante, si haces un cambio, solo tienes que hacerlo en un lugar.
- Dividir un programa largo en funciones te permite depurar las partes de una en una y luego ensamblarlas en un todo que funciona.
- Las funciones bien diseñadas suelen ser útiles para muchos programas. Una vez que escribes y depuras una, puedes reutilizarla.

3.9. Depuración

Depurar puede ser frustrante, pero también es desafiante, interesante y a veces incluso divertido. Y es una de las habilidades más importantes que puedes aprender.

En cierto modo, depurar es como hacer trabajo de detective. Se te dan pistas y tienes que inferir los eventos que llevaron a los resultados que ves.

Depurar también se parece a la ciencia experimental. Una vez que tienes una idea de qué va mal, modificas tu programa y vuelves a intentarlo. Si tu hipótesis era correcta, puedes predecir el resultado de la modificación, y das un paso más hacia un programa que funciona. Si tu hipótesis era incorrecta, tienes que proponer una nueva.

Para algunas personas, programar y depurar son lo mismo; es decir, programar es el proceso de depurar gradualmente un programa hasta que hace lo que quieres. La idea es que deberías empezar con un programa que funciona y hacer pequeñas modificaciones, depurándolas sobre la marcha.

Si te encuentras pasando mucho tiempo depurando, a menudo es una señal de que estás escribiendo demasiado código antes de empezar a hacer pruebas. Si das pasos más pequeños, puede que descubras que puedes avanzar más rápido.

3.10. Glosario

definición de función: Una sentencia que crea una función.

encabezado: La primera línea de una definición de función.

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

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

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

bucle: Una sentencia que ejecuta una o más sentencias, a menudo repetidamente.

variable local: Una variable definida dentro de una función, y a la que solo se puede acceder dentro de la función.

diagrama de pila: Una representación gráfica de un stack de funciones, sus variables y los valores a los que se refieren.

marco: Una caja en un diagrama de pila que representa una llamada a función. Contiene las variables locales y los parámetros de la función.

traceback: Una lista de las funciones que se están ejecutando, impresa cuando ocurre una excepción.

3.11. Ejercicios

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

3.11.1. Pregunta a un asistente virtual

Las sentencias dentro de una función o un bucle `for` se indentan con cuatro espacios, por convención. Pero no todo el mundo está de acuerdo con esa convención. Si tienes curiosidad por la historia de este gran debate, pregunta a un asistente virtual que te hable sobre "espacios y tabulaciones en Python".

Los asistentes virtuales son bastante buenos escribiendo funciones pequeñas.

1. Pregunta a tu asistente virtual favorito: "Escribe una función llamada repeat que reciba un string y un entero e imprima el string el número de veces indicado".
2. Si el resultado usa un bucle `for`, podrías preguntar: "¿Puedes hacerlo sin un bucle for?"
3. Elige cualquier otra función de este capítulo y pide a un asistente virtual que la escriba. El reto es describir la función con suficiente precisión para obtener lo que quieres. Usa el vocabulario que has aprendido hasta ahora en este libro.

Los asistentes virtuales también son bastante buenos depurando funciones.

1. Pregunta a un asistente virtual qué está mal en esta versión de `print_twice`.

```
def print_twice(string):
    print(cat)
    print(cat)
```

Y si te bloqueas en cualquiera de los ejercicios siguientes, considera pedir ayuda a un asistente virtual.

3.11.2. Ejercicio

Escribe una función llamada `print_right` que reciba un string llamado `text` como parámetro e imprima el string con suficientes espacios iniciales para que la última letra del string esté en la columna 40 de la pantalla.

Pista: usa la función `len`, el operador de concatenación de strings (`+`) y el operador de repetición de strings (`*`).

Aquí tienes un ejemplo que muestra cómo debería funcionar.

```
print_right("Monty")
print_right("Python's")
print_right("Flying Circus")
```

3.11.3. Ejercicio

Escribe una función llamada `triangle` que reciba un string y un entero, y dibuje una pirámide con la altura dada, hecha usando copias del string. Aquí tienes un ejemplo de una pirámide con `5` niveles, usando el string `'L'`.

```
triangle('L', 5)
```

3.11.4. Ejercicio

Escribe una función llamada `rectangle` que reciba un string y dos enteros, y dibuje un rectángulo con la anchura y altura dadas, hecho usando copias del string. Aquí tienes un ejemplo de un rectángulo con anchura `5` y altura `4`, hecho con el string `'H'`.

```
rectangle('H', 5, 4)
```

3.11.5. Ejercicio

La canción "99 Bottles of Beer" empieza con esta estrofa:

> 99 bottles of beer on the wall
> 99 bottles of beer
> Take one down, pass it around
> 98 bottles of beer on the wall

Luego la segunda estrofa es igual, excepto que empieza con 98 botellas y termina con 97. La canción continúa, durante muchísimo tiempo, hasta que hay 0 botellas de cerveza.

Escribe una función llamada `bottle_verse` que reciba un número como parámetro y muestre la estrofa que empieza con el número dado de botellas.

Pista: considera empezar con una función que pueda imprimir la primera, segunda o última línea de la estrofa, y luego úsala para escribir `bottle_verse`.

Usa esta llamada a función para mostrar la primera estrofa.

```
bottle_verse(99)
```

Si quieres imprimir la canción completa, puedes usar este bucle `for`, que cuenta hacia atrás desde `99` hasta `1`. No tienes que entender completamente este ejemplo; aprenderemos más sobre los bucles `for` y la función `range` más adelante.

```
for n in range(99, 0, -1):
    bottle_verse(n)
    print()
```

Think Python: 3.ª edición

Traducción al español por midudev (Miguel Ángel Durán).

Puedes comprar versiones impresas y ebook de *Think Python 3e* en Bookshop.org y Amazon.

4. Funciones e interfaces

Este capítulo introduce un módulo llamado `jupyturtle`, que te permite crear dibujos sencillos dando instrucciones a una turtle imaginaria. Usaremos este módulo para escribir funciones que dibujan cuadrados, polígonos y círculos, y para demostrar el **diseño de interfaces**, que es una forma de diseñar funciones que trabajan juntas.

4.1. El módulo jupyturtle

Para usar el módulo `jupyturtle`, podemos importarlo así.

```
import jupyturtle
```

Ahora podemos usar las funciones definidas en el módulo, como `make_turtle` y `forward`.

```
jupyturtle.make_turtle()
jupyturtle.forward(100)
```

`make_turtle` crea un **canvas**, que es un espacio en la pantalla donde podemos dibujar, y una turtle, que se representa con un caparazón circular y una cabeza triangular. El círculo muestra la

ubicación de la turtle y el triángulo indica la dirección hacia la que mira.

`forward` mueve la turtle una distancia dada en la dirección hacia la que mira, dibujando un segmento de línea por el camino. La distancia está en unidades arbitrarias: el tamaño real depende de la pantalla de tu ordenador.

Usaremos muchas veces funciones definidas en el módulo `jupyturtle`, así que estaría bien no tener que escribir el nombre del módulo cada vez. Eso es posible si importamos el módulo así.

```
from jupyturtle import make_turtle, forward
```

Esta versión de la sentencia `import` importa `make_turtle` y `forward` desde el módulo `jupyturtle`, de modo que podemos llamarlas así.

```
make_turtle()
forward(100)
```

`jupyturtle` proporciona otras dos funciones que usaremos, llamadas `left` y `right`. Las importaremos así.

```
from jupyturtle import left, right
```

`left` hace que la turtle gire a la izquierda. Recibe un argumento, que es el ángulo del giro en grados. Por ejemplo, podemos hacer un giro a la izquierda de 90 grados así.

```
make_turtle()
forward(50)
left(90)
forward(50)
```

Este programa mueve la turtle hacia el este y luego hacia el norte, dejando dos segmentos de línea detrás. Antes de continuar, prueba si puedes modificar el programa anterior para hacer un cuadrado.

4.2. Hacer un cuadrado

Aquí tienes una forma de hacer un cuadrado.

```
make_turtle()

forward(50)
left(90)

forward(50)
left(90)

forward(50)
left(90)

forward(50)
left(90)
```

Como este programa repite el mismo par de líneas cuatro veces, podemos hacer lo mismo de forma más concisa con un bucle `for`.

```
make_turtle()
for i in range(4):
    forward(50)
    left(90)
```

4.3. Encapsulación y generalización

Tomemos el código para dibujar cuadrados de la sección anterior y pongámoslo en una función llamada `square`.

```
def square():
    for i in range(4):
        forward(50)
        left(90)
```

Ahora podemos llamar a la función así.

```
make_turtle()
square()
```

Envolver un fragmento de código en una función se llama **encapsulación**. Uno de los beneficios de la encapsulación es que le da un nombre al código, que sirve como una especie de documentación. ¡Otra ventaja es que, si reutilizas el código, es más conciso llamar a una función dos veces que copiar y pegar el cuerpo!

En la versión actual, el tamaño del cuadrado siempre es `50`. Si queremos dibujar cuadrados de distintos tamaños, podemos tomar la longitud de los lados como parámetro.

```
def square(length):
    for i in range(4):
        forward(length)
        left(90)
```

Ahora podemos dibujar cuadrados de distintos tamaños.

```
make_turtle()
square(30)
square(60)
```

Añadir un parámetro a una función se llama **generalización** porque hace que la función sea más general: con la versión anterior, el cuadrado siempre tiene el mismo tamaño; con esta versión puede tener cualquier tamaño.

Si añadimos otro parámetro, podemos hacerla aún más general. La siguiente función dibuja polígonos regulares con un número dado de lados.

```
def polygon(n, length):
    angle = 360 / n
    for i in range(n):
        forward(length)
        left(angle)
```

En un polígono regular con `n` lados, el ángulo entre lados adyacentes es de `360 / n` grados.

El siguiente ejemplo dibuja un polígono de `7` lados con longitud de lado `30`.

```
make_turtle()
polygon(7, 30)
```

Cuando una función tiene más de unos pocos argumentos numéricos, es fácil olvidar qué son o en qué orden deberían ir. Puede ser una buena idea incluir los nombres de los parámetros en la lista de argumentos.

```
make_turtle()
polygon(n=7, length=30)
```

A veces se llaman "argumentos con nombre" porque incluyen los nombres de los parámetros. Pero en Python se llaman más a menudo **argumentos de palabra clave** (no deben confundirse con las palabras clave de Python como `for` y `def`).

Este uso del operador de asignación, `=`, sirve como recordatorio de cómo funcionan los argumentos y los parámetros: cuando llamas a una función, los argumentos se asignan a los parámetros.

4.4. Aproximar un círculo

Ahora supongamos que queremos dibujar un círculo. Podemos hacerlo, aproximadamente, dibujando un polígono con un gran número de lados, de modo que cada lado sea lo bastante pequeño como para que cueste verlo. Aquí tienes una función que usa `polygon` para dibujar un polígono de `30` lados que aproxima un círculo.

```
import math

def circle(radius):
    circumference = 2 * math.pi * radius
    n = 30
    length = circumference / n
    polygon(n, length)
```

`circle` recibe el radio del círculo como parámetro. Calcula `circumference`, que es la circunferencia de un círculo con el radio dado. `n` es el número de lados, así que `circumference / n` es la longitud de cada lado.

Esta función podría tardar mucho en ejecutarse. Podemos acelerarla llamando a `make_turtle` con un argumento de palabra clave llamado `delay` que establece el tiempo, en segundos, que la turtle espera después de cada paso. El valor predeterminado es `0.2` segundos; si lo establecemos en `0.02`, se ejecuta unas 10 veces más rápido.

```
make_turtle(delay=0.02)
circle(30)
```

Una limitación de esta solución es que `n` es una constante, lo que significa que para círculos muy grandes los lados son demasiado largos, y para círculos pequeños perdemos tiempo dibujando lados muy cortos. Una opción es generalizar la función tomando `n` como parámetro. Pero mantengámoslo sencillo por ahora.

4.5. Refactorización

Ahora escribamos una versión más general de `circle`, llamada `arc`, que recibe un segundo parámetro, `angle`, y dibuja un arco de un círculo que abarca el ángulo dado. Por ejemplo, si `angle` es `360` grados, dibuja un círculo completo. Si `angle` es `180` grados, dibuja un semicírculo.

Para escribir `circle`, pudimos reutilizar `polygon`, porque un polígono con muchos lados es una buena aproximación de un círculo. Pero no podemos usar `polygon` para escribir `arc`.

En su lugar, crearemos una versión más general de `polygon`, llamada `polyline`.

```
def polyline(n, length, angle):
    for i in range(n):
        forward(length)
        left(angle)
```

`polyline` recibe como parámetros el número de segmentos de línea que debe dibujar, `n`, la longitud de los segmentos, `length`, y el ángulo entre ellos, `angle`.

Ahora podemos reescribir `polygon` para que use `polyline`.

```
def polygon(n, length):
    angle = 360.0 / n
    polyline(n, length, angle)
```

Y podemos usar `polyline` para escribir `arc`.

```
def arc(radius, angle):
    arc_length = 2 * math.pi * radius * angle / 360
    n = 30
    length = arc_length / n
    step_angle = angle / n
    polyline(n, length, step_angle)
```

`arc` es similar a `circle`, salvo que calcula `arc_length`, que es una fracción de la circunferencia de un círculo.

Por último, podemos reescribir `circle` para que use `arc`.

```
def circle(radius):
    arc(radius,  360)
```

Para comprobar que estas funciones funcionan como esperamos, las usaremos para dibujar algo parecido a un caracol. Con `delay=0`, la turtle va lo más rápido posible.

```
make_turtle(delay=0)
polygon(n=20, length=9)
arc(radius=70, angle=70)
circle(radius=10)
```

En este ejemplo, empezamos con código que funcionaba y lo reorganizamos con funciones diferentes. Los cambios como este, que mejoran el código sin cambiar su comportamiento, se llaman **refactorización**.

Si lo hubiéramos planeado con antelación, quizá habríamos escrito `polyline` primero y evitado la refactorización, pero a menudo no sabes lo suficiente al principio de un proyecto para diseñar todas las funciones. Una vez que empiezas a programar, entiendes mejor el problema. A veces la refactorización es una señal de que has aprendido algo.

4.6. Diagrama de pila

Cuando llamamos a `circle`, llama a `arc`, que llama a `polyline`. Podemos usar un diagrama de pila para mostrar esta secuencia de llamadas a funciones y los parámetros de cada una.

```
from diagram import make_binding, make_frame, Frame, Stack

frame1 = make_frame(dict(radius=30), name='circle', loc='left')

frame2 = make_frame(dict(radius=30, angle=360), name='arc', loc='left', dx=1.1)

frame3 = make_frame(dict(n=60, length=3.04, angle=5.8),
                    name='polyline', loc='left', dx=1.1, offsetx=-0.27)

stack = Stack([frame1, frame2, frame3], dy=-0.4)
```

```
from diagram import diagram, adjust

width, height, x, y = [3.58, 1.31, 0.98, 1.06]
ax = diagram(width, height)
bbox = stack.draw(ax, x, y)
#adjust(x, y, bbox)
```

Observa que el valor de `angle` en `polyline` es diferente del valor de `angle` en `arc`. Los parámetros son locales, lo que significa que puedes usar el mismo nombre de parámetro en funciones diferentes; es una variable distinta en cada función y puede referirse a un valor diferente.

4.7. Un plan de desarrollo

Un **plan de desarrollo** es un proceso para escribir programas. El proceso que usamos en este capítulo es "encapsulación y generalización". Los pasos de este proceso son:

1. Empieza escribiendo un programa pequeño sin definiciones de funciones.
2. Una vez que consigas que el programa funcione, identifica una parte coherente, encapsula esa parte en una función y dale un nombre.
3. Generaliza la función añadiendo los parámetros adecuados.
4. Repite los pasos 1 a 3 hasta que tengas un conjunto de funciones que funcionen.
5. Busca oportunidades para mejorar el programa mediante refactorización. Por ejemplo, si tienes código similar en varios lugares, considera factorizarlo en una función adecuadamente general.

Este proceso tiene algunas desventajas (veremos alternativas más adelante), pero puede ser útil si no sabes de antemano cómo dividir el programa en funciones. Este enfoque te permite diseñar sobre la marcha.

El diseño de una función tiene dos partes:

- La **interfaz** es cómo se usa la función, incluyendo su nombre, los parámetros que recibe y lo que se supone que debe hacer.
- La **implementación** es cómo la función hace lo que se supone que debe hacer.

Por ejemplo, aquí tienes la primera versión de `circle` que escribimos, que usa `polygon`.

```
def circle(radius):
    circumference = 2 * math.pi * radius
    n = 30
    length = circumference / n
    polygon(n, length)
```

Y aquí tienes la versión refactorizada que usa `arc`.

```
def circle(radius):
    arc(radius,  360)
```

Estas dos funciones tienen la misma interfaz: reciben los mismos parámetros y hacen lo mismo, pero tienen implementaciones diferentes.

4.8. Docstrings

Un **docstring** es un string al principio de una función que explica la interfaz ("doc" es la abreviatura de "documentación"). Aquí tienes un ejemplo:

```
def polyline(n, length, angle):
    """Draws line segments with the given length and angle between them.

    n: integer number of line segments
    length: length of the line segments
    angle: angle between segments (in degrees)
    """
    for i in range(n):
        forward(length)
        left(angle)
```

Por convención, los docstrings son strings entre comillas triples, también conocidos como **strings multilínea** porque las comillas triples permiten que el string ocupe más de una línea.

Un docstring debería:

- Explicar de forma concisa qué hace la función, sin entrar en los detalles de cómo funciona,
- Explicar qué efecto tiene cada parámetro en el comportamiento de la función, e
- Indicar qué tipo debería tener cada parámetro, si no es obvio.

Escribir este tipo de documentación es una parte importante del diseño de interfaces. Una interfaz bien diseñada debería ser sencilla de explicar; si te cuesta explicar una de tus funciones, quizá la interfaz podría mejorarse.

4.9. Depuración

Una interfaz es como un contrato entre una función y quien la llama. Quien llama acepta proporcionar ciertos argumentos y la función acepta hacer cierto trabajo.

Por ejemplo, `polyline` requiere tres argumentos: `n` tiene que ser un entero; `length` debería ser un número positivo; y `angle` tiene que ser un número, que se entiende que está en grados.

Estos requisitos se llaman **precondiciones** porque se supone que deben ser verdaderos antes de que la función empiece a ejecutarse. Por el contrario, las condiciones al final de la función son **postcondiciones**. Las postcondiciones incluyen el efecto previsto de la función (como dibujar segmentos de línea) y cualquier efecto secundario (como mover la turtle o hacer otros cambios).

Las precondiciones son responsabilidad de quien llama. Si quien llama viola una precondición y la función no funciona correctamente, el bug está en quien llama, no en la función.

Si las precondiciones se satisfacen y las postcondiciones no, el bug está en la función. Si tus precondiciones y postcondiciones están claras, pueden ayudar con la depuración.

4.10. Glosario

diseño de interfaces: Un proceso para diseñar la interfaz de una función, que incluye los parámetros que debería recibir.

lienzo: Una ventana usada para mostrar elementos gráficos, incluyendo líneas, círculos, rectángulos y otras formas.

encapsulación: El proceso de transformar una secuencia de sentencias en una definición de función.

generalización: El proceso de reemplazar algo innecesariamente específico (como un número) por algo adecuadamente general (como una variable o un parámetro).

argumento de palabra clave: Un argumento que incluye el nombre del parámetro.

refactorización: El proceso de modificar un programa que funciona para mejorar las interfaces de las funciones y otras cualidades del código.

plan de desarrollo: Un proceso para escribir programas.

docstring: Un string que aparece al principio de una definición de función para documentar la interfaz de la función.

string multilínea: Un string encerrado entre comillas triples que puede ocupar más de una línea de un programa.

precondición: Un requisito que debería satisfacer quien llama antes de que empiece una función.

postcondición: Un requisito que debería satisfacer la función antes de terminar.

4.11. Ejercicios

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

Para los ejercicios siguientes, hay algunas funciones más de turtle que quizá quieras usar.

- `penup` levanta el lápiz imaginario de la turtle para que no deje rastro cuando se mueve.
- `pendown` vuelve a bajar el lápiz.

La siguiente función usa `penup` y `pendown` para mover la turtle sin dejar rastro.

```
from jupyturtle import penup, pendown

def jump(length):
    """Move forward length units without leaving a trail.

    Postcondition: Leaves the pen down.
    """
    penup()
    forward(length)
    pendown()
```

4.11.1. Ejercicio

Escribe una función llamada `rectangle` que dibuje un rectángulo con longitudes de lado dadas. Por ejemplo, aquí tienes un rectángulo de `80` unidades de ancho y `40` unidades de alto.

Puedes usar el siguiente código para probar tu función.

```
make_turtle()
rectangle(80, 40)
```

4.11.2. Ejercicio

Escribe una función llamada `rhombus` que dibuje un rombo con una longitud de lado dada y un ángulo interior dado. Por ejemplo, aquí tienes un rombo con longitud de lado `50` y un ángulo interior de `60` grados.

Puedes usar el siguiente código para probar tu función.

```
make_turtle()
rhombus(50, 60)
```

4.11.3. Ejercicio

Ahora escribe una función más general llamada `parallelogram` que dibuje un cuadrilátero con lados paralelos. Luego reescribe `rectangle` y `rhombus` para que usen `parallelogram`.

Puedes usar el siguiente código para probar tus funciones.

```
make_turtle(width=400)
jump(-120)

rectangle(80, 40)
jump(100)
rhombus(50, 60)
jump(80)
parallelogram(80, 50, 60)
```

4.11.4. Ejercicio

Escribe un conjunto de funciones adecuadamente general que pueda dibujar formas como esta.

Pista: escribe una función llamada `triangle` que dibuje un segmento triangular, y luego una función llamada `draw_pie` que use `triangle`.

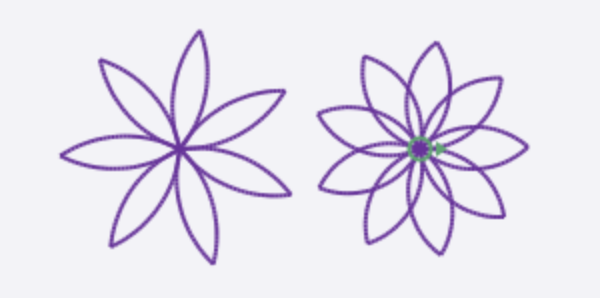
Puedes usar el siguiente código para probar tus funciones.

```
turtle = make_turtle(delay=0)
jump(-80)

size = 40
draw_pie(5, size)
jump(2*size)
draw_pie(6, size)
jump(2*size)
draw_pie(7, size)
```

4.11.5. Ejercicio

Escribe un conjunto de funciones adecuadamente general que pueda dibujar flores como esta.



Pista: usa `arc` para escribir una función llamada `petal` que dibuje un pétalo de flor.

Puedes usar el siguiente código para probar tus funciones.

Como la solución dibuja muchos segmentos de línea pequeños, tiende a ralentizarse mientras se ejecuta. Para evitarlo, puedes añadir el argumento de palabra clave `auto_render=False` para evitar dibujar después de cada paso, y luego llamar a la función `render` al final para mostrar el resultado.

Mientras estés depurando, quizá quieras quitar `auto_render=False`.

```
from jupyturtle import render

turtle = make_turtle(auto_render=False)

jump(-60)
n = 7
radius = 60
angle = 60
flower(n, radius, angle)

jump(120)
n = 9
radius = 40
angle = 85
flower(n, radius, angle)

render()
```

4.11.6. Pregunta a un asistente virtual

Hay varios módulos como `jupyturtle` en Python, y el que usamos en este capítulo se ha personalizado para este libro. Así que, si pides ayuda a un asistente virtual, no sabrá qué módulo usar. Pero si le das algunos ejemplos con los que trabajar, probablemente pueda deducirlo. Por ejemplo, prueba este prompt y mira si puede escribir una función que dibuje una espiral:

```
The following program uses a turtle graphics module to draw a circle:

from jupyturtle import make_turtle, forward, left
import math

def polygon(n, length):
    angle = 360 / n
    for i in range(n):
        forward(length)
        left(angle)

def circle(radius):
    circumference = 2 * math.pi * radius
    n = 30
    length = circumference / n
    polygon(n, length)

make_turtle(delay=0)
circle(30)

Write a function that draws a spiral.
```

Ten en cuenta que el resultado podría usar características que aún no hemos visto, y podría tener errores. Copia el código del asistente virtual y comprueba si puedes hacerlo funcionar. Si no obtuviste lo que querías, prueba a modificar el prompt.

Think Python: 3.ª edición

Traducción al español por midudev (Miguel Ángel Durán).

Puedes comprar versiones impresas y ebook de *Think Python 3e* en Bookshop.org y Amazon.

5. Condicionales y recursión

El tema principal de este capítulo es la sentencia `if`, que ejecuta código distinto según el estado del programa. Y con la sentencia `if` podremos explorar una de las ideas más potentes de la informática: la **recursión**.

Pero empezaremos con tres características nuevas: el operador módulo, las expresiones booleanas y los operadores lógicos.

5.1. División entera y módulo

Recuerda que el operador de división entera, `//`, divide dos números y redondea hacia abajo hasta un entero. Por ejemplo, supón que la duración de una película es de 105 minutos. Puede que quieras saber cuánto es eso en horas. La división convencional devuelve un número de punto flotante:

```
minutes = 105
minutes / 60
```

Pero normalmente no escribimos las horas con decimales. La división entera devuelve el número entero de horas, redondeando hacia abajo:

```
minutes = 105
hours = minutes // 60
hours
```

Para obtener el resto, podrías restar una hora expresada en minutos:

```
remainder = minutes - hours * 60
remainder
```

O podrías usar el **operador módulo**, `%`, que divide dos números y devuelve el resto.

```
remainder = minutes % 60
remainder
```

El operador módulo es más útil de lo que podría parecer. Por ejemplo, puede comprobar si un número es divisible por otro: si `x % y` es cero, entonces `x` es divisible por `y`.

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

```
x = 123
x % 10
```

```
x % 100
```

Por último, el operador módulo puede hacer «aritmética de reloj». Por ejemplo, si un evento empieza a las 11 AM y dura tres horas, podemos usar el operador módulo para averiguar a qué hora termina.

```
start = 11
duration = 3
end = (start + duration) % 12
end
```

El evento terminaría a las 2 PM.

5.2. Expresiones booleanas

Una **expresión booleana** es una expresión que es verdadera o falsa. Por ejemplo, las siguientes expresiones usan el operador de igualdad, `==`, que compara dos valores y produce `True` si son iguales y `False` en caso contrario:

```
5 == 5
```

```
5 == 7
```

Un error común es usar un solo signo igual (`=`) en lugar de un doble signo igual (`==`). Recuerda que `=` asigna un valor a una variable y `==` compara dos valores.

```
x = 5
y = 7
```

```
x == y
```

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

```
type(True)
```

```
type(False)
```

El operador `==` es uno de los **operadores relacionales**; los otros son:

```
x != y               # x is not equal to y
```

```
x > y                # x is greater than y
```

```
x < y               # x is less than to y
```

```
x >= y               # x is greater than or equal to y
```

```
x <= y               # x is less than or equal to y
```

5.3. Operadores lógicos

Para combinar valores booleanos en expresiones, podemos usar **operadores lógicos**. Los más comunes son `and`, `or` y `not`. El significado de estos operadores es parecido a su significado en inglés. Por ejemplo, el valor de la siguiente expresión es `True` solo si `x` es mayor que `0` *y* menor que `10`.

```
x > 0 and x < 10
```

La siguiente expresión es `True` si *una o ambas* condiciones son verdaderas, es decir, si el número es divisible por 2 *o* por 3:

```
x % 2 == 0 or x % 3 == 0
```

Por último, el operador `not` niega una expresión booleana, de modo que la siguiente expresión es `True` si `x > y` es `False`.

```
not x > y
```

En sentido estricto, los operandos de un operador lógico deberían ser expresiones booleanas, pero Python no es muy estricto. Cualquier número distinto de cero se interpreta como `True`:

```
42 and True
```

Esta flexibilidad puede ser útil, pero tiene algunas sutilezas que pueden resultar confusas. Quizá prefieras evitarla.

5.4. Sentencias `if`

Para escribir programas útiles, casi siempre necesitamos la capacidad de comprobar condiciones y cambiar el comportamiento del programa en consecuencia. Las **sentencias condicionales** nos dan esta capacidad. La forma más sencilla es la sentencia `if`:

```
if x > 0:
    print('x is positive')
```

`if` es una palabra clave de Python. Las sentencias `if` tienen la misma estructura que las definiciones de función: una cabecera seguida de una sentencia indentada o de una secuencia de sentencias llamada **bloque**.

La expresión booleana que va después de `if` se llama **condición**. Si es verdadera, se ejecutan las sentencias del bloque indentado. Si no, no se ejecutan.

No hay límite para el número de sentencias que pueden aparecer en el bloque, pero tiene que haber al menos una. A veces resulta útil tener un bloque que no hace nada, normalmente como marcador de posición para código que todavía no has escrito. En ese caso, puedes usar la sentencia `pass`, que no hace nada.

```
if x < 0:
    pass          # TODO: need to handle negative values!
```

La palabra `TODO` en un comentario es un recordatorio convencional de que hay algo que tienes que hacer más adelante.

5.5. La cláusula `else`

Una sentencia `if` puede tener una segunda parte, llamada cláusula `else`. La sintaxis tiene este aspecto:

```
if x % 2 == 0:
    print('x is even')
else:
    print('x is odd')
```

Si la condición es verdadera, se ejecuta la primera sentencia indentada; si no, se ejecuta la segunda sentencia indentada.

En este ejemplo, si `x` es par, el resto al dividir `x` entre `2` es `0`, así que la condición es verdadera y el programa muestra `x is even`. Si `x` es impar, el resto es `1`, así que la condición es falsa, y el programa muestra `x is odd`.

Como la condición debe ser verdadera o falsa, se ejecutará exactamente una de las alternativas. Las alternativas se llaman **ramas**.

5.6. Condicionales encadenados

A veces hay más de dos posibilidades y necesitamos más de dos ramas. Una forma de expresar un cálculo así es un **condicional encadenado**, que incluye una cláusula `elif`.

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

`elif` es una abreviatura de «else if». No hay límite en el número de cláusulas `elif`. Si hay una cláusula `else`, tiene que ir al final, pero no tiene por qué haber una.

Cada condición se comprueba en orden. Si la primera es falsa, se comprueba la siguiente, y así sucesivamente. Si una de ellas es verdadera, se ejecuta la rama correspondiente y termina la sentencia `if`. Aunque más de una condición sea verdadera, solo se ejecuta la primera rama verdadera.

5.7. Condicionales anidados

Un condicional también puede estar anidado dentro de otro. Podríamos haber escrito el ejemplo de la sección anterior así:

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

La sentencia `if` exterior contiene dos ramas. La primera rama contiene una sentencia sencilla. La segunda rama contiene otra sentencia `if`, que tiene dos ramas propias. Esas dos ramas son ambas sentencias sencillas, aunque también podrían haber sido sentencias condicionales.

Aunque la indentación de las sentencias hace visible la estructura, los **condicionales anidados** pueden ser difíciles de leer. Te sugiero que los evites cuando puedas.

Los operadores lógicos suelen proporcionar una forma de simplificar sentencias condicionales anidadas. Aquí tienes un ejemplo con un condicional anidado.

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

La sentencia `print` se ejecuta solo si superamos ambos condicionales, así que conseguimos el mismo efecto con el operador `and`.

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

Para este tipo de condición, Python ofrece una opción más concisa:

```
if 0 < x < 10:
    print('x is a positive single-digit number.')
```

5.8. Recursión

Es legal que una función se llame a sí misma. Puede que no sea obvio por qué eso es algo bueno, pero resulta ser una de las cosas más mágicas que puede hacer un programa. Aquí tienes un ejemplo.

```
def countdown(n):
    if n <= 0:
        print('Blastoff!')
    else:
        print(n)
        countdown(n-1)
```

Si `n` es 0 o negativo, `countdown` muestra la palabra «Blastoff!». En caso contrario, muestra `n` y luego se llama a sí misma, pasando `n-1` como argumento.

Esto es lo que ocurre cuando llamamos a esta función con el argumento `3`.

```
countdown(3)
```

La ejecución de `countdown` empieza con `n=3`, y como `n` es mayor que `0`, muestra `3` y luego se llama a sí misma....

> La ejecución de `countdown` empieza con `n=2`, y como `n` es mayor que `0`, muestra `2` y luego se llama a sí misma....
>
> > La ejecución de `countdown` empieza con `n=1`, y como `n` es mayor que `0`, muestra `1` y luego se llama a sí misma....

> > La ejecución de `countdown` empieza con `n=0`, y como `n` no es mayor que `0`, muestra «Blastoff!» y retorna.
>
> La `countdown` que recibió `n=1` retorna.

La `countdown` que recibió `n=2` retorna.

La `countdown` que recibió `n=3` retorna.

Una función que se llama a sí misma es **recursiva**. Como otro ejemplo, podemos escribir una función que imprime una cadena `n` veces.

```
def print_n_times(string, n):
    if n > 0:
        print(string)
        print_n_times(string, n-1)
```

Si `n` es positivo, `print_n_times` muestra el valor de `string` y luego se llama a sí misma, pasando `string` y `n-1` como argumentos.

Si `n` es `0` o negativo, la condición es falsa y `print_n_times` no hace nada.

Así es como funciona.

```
print_n_times('Spam ', 4)
```

Para ejemplos sencillos como este, probablemente sea más fácil usar un bucle `for`. Pero más adelante veremos ejemplos que son difíciles de escribir con un bucle `for` y fáciles de escribir con recursión, así que conviene empezar pronto.

5.9. Diagramas de stack para funciones recursivas

Aquí tienes un diagrama de stack que muestra los marcos creados cuando llamamos a `countdown` con `n = 3`.

```
from diagram import make_frame, Stack

frames = []
for n in [3,2,1,0]:
    d = dict(n=n)
    frame = make_frame(d, name='countdown', dy=-0.3, loc='left')
    frames.append(frame)

stack = Stack(frames, dy=-0.5)
```

```
from diagram import diagram, adjust


width, height, x, y = [1.74, 2.04, 1.05, 1.77]
ax = diagram(width, height)
bbox = stack.draw(ax, x, y)
# adjust(x, y, bbox)
```

Los cuatro marcos de `countdown` tienen valores distintos para el parámetro `n`. La parte inferior del stack, donde `n=0`, se llama **caso base**. No hace una llamada recursiva, así que no hay más marcos.

```
from diagram import make_frame, Stack
from diagram import diagram, adjust

frames = []
for n in [2,1,0]:
    d = dict(string='Hello', n=n)
    frame = make_frame(d, name='print_n_times', dx=1.3, loc='left')
    frames.append(frame)

stack = Stack(frames, dy=-0.5)

width, height, x, y = [3.53, 1.54, 1.54, 1.27]
ax = diagram(width, height)
bbox = stack.draw(ax, x, y)
# adjust(x, y, bbox)
```

5.10. Recursión infinita

Si una recursión nunca alcanza un caso base, sigue haciendo llamadas recursivas para siempre, y el programa nunca termina. Esto se conoce como **recursión infinita**, y por lo general no es una buena idea. Aquí tienes una función mínima con una recursión infinita.

```
def recurse():
    recurse()
```

Cada vez que se llama a `recurse`, se llama a sí misma, lo que crea otro marco. En Python, hay un límite para el número de marcos que pueden estar en el stack al mismo tiempo. Si un programa supera el límite, provoca un error en tiempo de ejecución.

```
%xmode Context
```

```
recurse()
```

El traceback indica que había casi 3000 marcos en el stack cuando ocurrió el error.

Si te encuentras por accidente con una recursión infinita, revisa tu función para confirmar que hay un caso base que no hace una llamada recursiva. Y si hay un caso base, comprueba si tienes garantizado llegar a él.

5.11. Entrada por teclado

Los programas que hemos escrito hasta ahora no aceptan entrada del usuario. Simplemente hacen lo mismo cada vez.

Python proporciona una función integrada llamada `input` que detiene el programa y espera a que el usuario escriba algo. Cuando el usuario pulsa *Return* o *Enter*, el programa continúa y `input` devuelve lo que el usuario ha escrito como una cadena.

```
text = input()
```

Antes de obtener entrada del usuario, puede que quieras mostrar un prompt que le indique qué debe escribir. `input` puede recibir un prompt como argumento:

```
name = input('What...is your name?\n')
name
```

La secuencia `\n` al final del prompt representa una **nueva línea**, que es un carácter especial que provoca un salto de línea; así, la entrada del usuario aparece debajo del prompt.

Si esperas que el usuario escriba un entero, puedes usar la función `int` para convertir el valor de retorno a `int`.

```
prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
speed = input(prompt)
speed
```

Pero si escribe algo que no es un entero, obtendrás un error en tiempo de ejecución.

```
%xmode Minimal
```

```
int(speed)
```

Más adelante veremos cómo manejar este tipo de error.

5.12. Debugging

Cuando ocurre un error de sintaxis o en tiempo de ejecución, el mensaje de error contiene mucha información, pero puede resultar abrumador. Las partes más útiles suelen ser:

- Qué tipo de error fue, y
- Dónde ocurrió.

Los errores de sintaxis suelen ser fáciles de encontrar, pero hay algunas trampas. Los errores relacionados con espacios y tabulaciones pueden ser complicados porque son invisibles y estamos acostumbrados a ignorarlos.

```
x = 5
 y = 6
```

En este ejemplo, el problema es que la segunda línea está indentada con un espacio. Pero el mensaje de error señala a `y`, lo que resulta engañoso. Los mensajes de error indican dónde se descubrió el problema, pero el error real podría estar antes en el código.

Lo mismo ocurre con los errores en tiempo de ejecución. Por ejemplo, supón que intentas convertir una razón a decibelios, así:

```
%xmode Context
```

```
import math
numerator = 9
denominator = 10
ratio = numerator // denominator
decibels = 10 * math.log10(ratio)
```

El mensaje de error indica la línea 5, pero no hay nada malo en esa línea. El problema está en la línea 4, que usa división entera en lugar de división de punto flotante; como resultado, el valor de `ratio` es `0`. Cuando llamamos a `math.log10`, obtenemos un `ValueError` con el mensaje `math domain error`, porque `0` no está en el «dominio» de argumentos válidos para `math.log10`, ya que el logaritmo de `0` no está definido.

En general, deberías tomarte el tiempo de leer cuidadosamente los mensajes de error, pero no asumas que todo lo que dicen es correcto.

5.13. Glosario

recursión: El proceso de llamar a la función que se está ejecutando en ese momento.

operador módulo: Un operador, `%`, que funciona con enteros y devuelve el resto cuando un número se divide entre otro.

expresión booleana: Una expresión cuyo valor es `True` o `False`.

operador relacional: Uno de los operadores que compara sus operandos: `==`, `!=`, `>`, `<`, `>=` y `<=`.

operador lógico: Uno de los operadores que combina expresiones booleanas, incluidos `and`, `or` y `not`.

sentencia condicional: Una sentencia que controla el flujo de ejecución dependiendo de alguna condición.

condición: La expresión booleana en una sentencia condicional que determina qué rama se ejecuta.

bloque: Una o más sentencias indentadas para indicar que forman parte de otra sentencia.

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

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

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

recursiva: Una función que se llama a sí misma es recursiva.

caso base: Una rama condicional en una función recursiva que no hace una llamada recursiva.

recursión infinita: Una recursión que no tiene caso base o que nunca lo alcanza. Con el tiempo, una recursión infinita provoca un error en tiempo de ejecución.

nueva línea: Un carácter que crea un salto de línea entre dos partes de una cadena.

5.14. Ejercicios

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

5.14.1. Pregunta a un asistente virtual

- Pregunta a un asistente virtual: «¿Cuáles son algunos usos del operador módulo?»
- Python proporciona operadores para calcular las operaciones lógicas `and`, `or` y `not`, pero no tiene un operador que calcule la operación `or` exclusiva, que normalmente se escribe `xor`. Pregunta a un asistente: «¿Qué es la operación lógica xor y cómo la calculo en Python?»

En este capítulo, vimos dos formas de escribir una sentencia `if` con tres ramas: usando un condicional encadenado o un condicional anidado. Puedes usar un asistente virtual para convertir

de una forma a la otra. Por ejemplo, pregunta a un asistente virtual: «Convierte esta sentencia en un condicional encadenado».

```
x = 5
y = 7
```

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

Pregunta a un asistente virtual: «Reescribe esta sentencia con un único condicional».

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

Comprueba si un asistente virtual puede simplificar esta complejidad innecesaria.

```
if not x <= 0 and not x >= 10:
    print('x is a positive single-digit number.')
```

Aquí tienes un intento de función recursiva que cuenta hacia atrás de dos en dos.

```
def countdown_by_two(n):
    if n == 0:
        print('Blastoff!')
    else:
        print(n)
        countdown_by_two(n-2)
```

Parece funcionar.

```
countdown_by_two(6)
```

Pero tiene un error. Pregunta a un asistente virtual qué está mal y cómo arreglarlo. Pega aquí la solución que te proporcione y pruébala.

5.14.2. Ejercicio

El módulo `time` proporciona una función, también llamada `time`, que devuelve el número de segundos transcurridos desde la «época Unix», que es el 1 de enero de 1970, 00:00:00 UTC (tiempo universal coordinado).

```
from time import time

now = time()
now
```

Usa la división entera y el operador módulo para calcular el número de días transcurridos desde el 1 de enero de 1970 y la hora actual del día en horas, minutos y segundos.

Puedes leer más sobre el módulo `time` en https://docs.python.org/3/library/time.html.

5.14.3. Ejercicio

Si te dan tres palos, puede que puedas colocarlos formando un triángulo, o puede que no. Por ejemplo, si uno de los palos mide 12 pulgadas y los otros dos miden una pulgada, no podrás hacer que los palos cortos se encuentren en el centro. Para tres longitudes cualesquiera, hay una prueba para ver si es posible formar un triángulo:

> Si cualquiera de las tres longitudes es mayor que la suma de las otras dos, entonces no puedes formar un triángulo. En caso contrario, sí puedes. (Si la suma de dos longitudes es igual a la tercera, forman lo que se llama un triángulo «degenerado».)

Escribe una función llamada `is_triangle` que reciba tres enteros como argumentos, y que imprima «Yes» o «No», dependiendo de si se puede o no formar un triángulo con palos de las longitudes dadas. Pista: usa un condicional encadenado.

Prueba tu función con los siguientes casos.

```
is_triangle(4, 5, 6)   # should be Yes
```

```
is_triangle(1, 2, 3)   # should be Yes
```

```
is_triangle(6, 2, 3)   # should be No
```

```
is_triangle(1, 1, 12)   # should be No
```

5.14.4. Ejercicio

¿Cuál es la salida del siguiente programa? Dibuja un diagrama de stack que muestre el estado del programa cuando imprime el resultado.

```
def recurse(n, s):
    if n == 0:
        print(s)
    else:
        recurse(n-1, n+s)

recurse(3, 0)
```

5.14.5. Ejercicio

Los siguientes ejercicios usan el módulo `jupyturtle`, descrito en el Capítulo 4.

Lee la siguiente función y mira si puedes averiguar qué hace. Luego ejecútala y comprueba si acertaste. Ajusta los valores de `length`, `angle` y `factor`, y observa qué efecto tienen en el resultado. Si no tienes claro cómo funciona, prueba a preguntar a un asistente virtual.

```
from jupyturtle import forward, left, right, back

def draw(length):
    angle = 50
    factor = 0.6

    if length > 5:
        forward(length)
        left(angle)
        draw(factor * length)
        right(2 * angle)
        draw(factor * length)
        left(angle)
        back(length)
```

5.14.6. Ejercicio

Pregunta a un asistente virtual: «¿Qué es la curva de Koch?»

Para dibujar una curva de Koch con longitud `x`, todo lo que tienes que hacer es

1. Dibujar una curva de Koch con longitud `x/3`.
2. Girar a la izquierda 60 grados.
3. Dibujar una curva de Koch con longitud `x/3`.
4. Girar a la derecha 120 grados.
5. Dibujar una curva de Koch con longitud `x/3`.
6. Girar a la izquierda 60 grados.
7. Dibujar una curva de Koch con longitud `x/3`.

La excepción es si `x` es menor que `5`; en ese caso, puedes dibujar simplemente una línea recta con longitud `x`.

Escribe una función llamada `koch` que reciba `x` como argumento y dibuje una curva de Koch con la longitud dada.

El resultado debería verse así:

```
make_turtle(delay=0)
koch(120)
```

Cuando tengas `koch` funcionando, puedes usar este bucle para dibujar tres curvas de Koch con forma de copo de nieve.

```
make_turtle(delay=0, height=300)
for i in range(3):
    koch(120)
    right(120)
```

5.14.7. Ejercicio

Los asistentes virtuales conocen las funciones del módulo `jupyturtle`, pero hay muchas versiones de estas funciones, con nombres distintos, así que un asistente virtual podría no saber de cuál estás hablando.

Para resolver este problema, puedes proporcionar información adicional antes de hacer una pregunta. Por ejemplo, podrías empezar un prompt con «Aquí tienes un programa que usa el módulo `jupyturtle`», y luego pegar uno de los ejemplos de este capítulo. Después de eso, el asistente virtual debería poder generar código que use este módulo.

Como ejemplo, pide a un asistente virtual un programa que dibuje un triángulo de Sierpiński. El código que obtengas debería ser un buen punto de partida, pero quizá tengas que hacer algo de debugging. Si el primer intento no funciona, puedes contarle al asistente virtual qué ocurrió y pedir ayuda, o puedes depurarlo por tu cuenta.

Este es un posible aspecto del resultado, aunque la versión que obtengas podría ser diferente.

```
make_turtle(delay=0, height=200)

draw_sierpinski(100, 3)
```

Puedes comprar las versiones impresa y ebook de *Think Python 3e* en Bookshop.org y Amazon.

6. Valores de retorno

En capítulos anteriores, hemos usado funciones integradas – como `abs` y `round` – y funciones del módulo math – como `sqrt` y `pow`. Cuando llamas a una de estas funciones, devuelve un valor que puedes asignar a una variable o usar como parte de una expresión.

Las funciones que hemos escrito hasta ahora son diferentes. Algunas usan la función `print` para mostrar valores, y otras usan funciones de turtle para dibujar figuras. Pero no devuelven valores que asignemos a variables o usemos en expresiones.

En este capítulo veremos cómo escribir funciones que devuelven valores.

6.1. Algunas funciones tienen valores de retorno

Cuando llamas a una función como `math.sqrt`, el resultado se llama **valor de retorno**. Si la llamada a la función aparece al final de una celda, Jupyter muestra inmediatamente el valor de retorno.

```
import math

math.sqrt(42 / math.pi)
```

Si asignas el valor de retorno a una variable, no se muestra.

```
radius = math.sqrt(42 / math.pi)
```

Pero puedes mostrarlo más tarde.

```
radius
```

O puedes usar el valor de retorno como parte de una expresión.

```
radius + math.sqrt(42 / math.pi)
```

Aquí tienes un ejemplo de una función que devuelve un valor.

```
def circle_area(radius):
    area = math.pi * radius**2
    return area
```

`circle_area` toma `radius` como parámetro y calcula el área de un círculo con ese radio.

La última línea es una sentencia `return` que devuelve el valor de `area`.

Si llamamos a la función así, Jupyter muestra el valor de retorno.

```
circle_area(radius)
```

Podemos asignar el valor de retorno a una variable.

```
a = circle_area(radius)
```

O usarlo como parte de una expresión.

```
circle_area(radius) + 2 * circle_area(radius / 2)
```

Más tarde podemos mostrar el valor de la variable a la que asignamos el resultado.

```
a
```

Pero no podemos acceder a `area`.

```
area
```

`area` es una variable local dentro de una función, así que no podemos acceder a ella desde fuera de la función.

6.2. Y algunas tienen None

Si una función no tiene una sentencia `return`, devuelve `None`, que es un valor especial como `True` y `False`. Por ejemplo, aquí está la función `repeat` del Capítulo 3.

```
def repeat(word, n):
    print(word * n)
```

Si la llamamos así, muestra la primera línea de la canción "Finland" de Monty Python.

```
repeat('Finland, ', 3)
```

Esta función usa la función `print` para mostrar una cadena, pero no usa una sentencia `return` para devolver un valor. Si asignamos el resultado a una variable, muestra la cadena de todos modos.

```
result = repeat('Finland, ', 3)
```

Y si mostramos el valor de la variable, no obtenemos nada.

```
result
```

`result` en realidad tiene un valor, pero Jupyter no lo muestra. Sin embargo, podemos mostrarlo así.

```
print(result)
```

El valor de retorno de `repeat` es `None`.

Ahora aquí tienes una función parecida a `repeat`, excepto que sí tiene un valor de retorno.

```
def repeat_string(word, n):
    return word * n
```

Observa que podemos usar una expresión en una sentencia `return`, no solo una variable.

Con esta versión, podemos asignar el resultado a una variable. Cuando la función se ejecuta, no muestra nada.

```
line = repeat_string('Spam, ', 4)
```

Pero más tarde podemos mostrar el valor asignado a `line`.

```
line
```

Una función como esta se llama **función pura** porque no muestra nada ni tiene ningún otro efecto – aparte de devolver un valor.

6.3. Valores de retorno y condicionales

Si Python no proporcionara `abs`, podríamos escribirla así.

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

Si `x` es negativo, la primera sentencia `return` devuelve `-x` y la función termina inmediatamente. En caso contrario, la segunda sentencia `return` devuelve `x` y la función termina. Así que esta función es correcta.

Sin embargo, si pones sentencias `return` en un condicional, tienes que asegurarte de que todos los caminos posibles del programa lleguen a una sentencia `return`. Por ejemplo, aquí tienes una versión incorrecta de `absolute_value`.

```
def absolute_value_wrong(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```

Esto es lo que ocurre si llamamos a esta función con `0` como argumento.

```
absolute_value_wrong(0)
```

¡No obtenemos nada! Este es el problema: cuando `x` es `0`, ninguna condición es verdadera, y la función termina sin llegar a una sentencia `return`, lo que significa que el valor de retorno es `None`, así que Jupyter no muestra nada.

Como otro ejemplo, aquí tienes una versión de `absolute_value` con una sentencia `return` adicional al final.

```
def absolute_value_extra_return(x):
    if x < 0:
        return -x
    else:
        return x

    return 'This is dead code'
```

Si `x` es negativo, se ejecuta la primera sentencia `return` y la función termina. En caso contrario, se ejecuta la segunda sentencia `return` y la función termina. De cualquier manera, nunca llegamos a la tercera sentencia `return` – así que nunca puede ejecutarse.

El código que nunca puede ejecutarse se llama **código muerto**. En general, el código muerto no hace daño, pero a menudo indica un malentendido y podría confundir a alguien que intenta entender el programa.

6.4. Desarrollo incremental

A medida que escribas funciones más grandes, puede que notes que pasas más tiempo depurando. Para enfrentarte a programas cada vez más complejos, quizá quieras probar el **desarrollo incremental**, que es una forma de añadir y probar solo una pequeña cantidad de código cada vez.

Como ejemplo, supón que quieres encontrar la distancia entre dos puntos representados por las coordenadas (x_1, y_1) y (x_2, y_2). Por el teorema de Pitágoras, la distancia es:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

El primer paso es considerar cómo debería ser una función `distance` en Python – es decir, cuáles son las entradas (parámetros) y cuál es la salida (valor de retorno).

Para esta función, las entradas son las coordenadas de los puntos. El valor de retorno es la distancia. De inmediato puedes escribir un esquema de la función:

```
def distance(x1, y1, x2, y2):
    return 0.0
```

Esta versión todavía no calcula distancias – siempre devuelve cero. Pero es una función completa con un valor de retorno, lo que significa que puedes probarla antes de hacerla más complicada.

Para probar la nueva función, la llamaremos con argumentos de ejemplo:

```
distance(1, 2, 4, 6)
```

Elegí estos valores para que la distancia horizontal sea `3` y la distancia vertical sea `4`. Así, el resultado es `5`, la hipotenusa de un triángulo rectángulo `3-4-5`. Al probar una función, es útil conocer la respuesta correcta.

En este punto hemos confirmado que la función se ejecuta y devuelve un valor, y podemos empezar a añadir código al cuerpo. Un buen siguiente paso es encontrar las diferencias `x2 - x1` e `y2 - y1`. Aquí tienes una versión que almacena esos valores en variables temporales y los muestra.

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print('dx is', dx)
    print('dy is', dy)
    return 0.0
```

Si la función funciona, debería mostrar `dx is 3` y `dy is 4`. Si es así, sabemos que la función está recibiendo los argumentos correctos y realizando correctamente el primer cálculo. Si no, solo hay unas pocas líneas que revisar.

```
distance(1, 2, 4, 6)
```

Hasta aquí, bien. A continuación calculamos la suma de los cuadrados de `dx` e `dy`:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print('dsquared is: ', dsquared)
    return 0.0
```

De nuevo, podemos ejecutar la función y comprobar la salida, que debería ser `25`.

```
distance(1, 2, 4, 6)
```

Por último, podemos usar `math.sqrt` para calcular la distancia:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    print("result is", result)
```

Y probarla.

```
distance(1, 2, 4, 6)
```

El resultado es correcto, pero esta versión de la función muestra el resultado en lugar de devolverlo, así que el valor de retorno es `None`.

Podemos arreglarlo sustituyendo la función `print` por una sentencia `return`.

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result
```

Esta versión de `distance` es una función pura. Si la llamamos así, solo se muestra el resultado.

```
distance(1, 2, 4, 6)
```

Y si asignamos el resultado a una variable, no se muestra nada.

```
d = distance(1, 2, 4, 6)
```

Las sentencias `print` que escribimos son útiles para depurar, pero una vez que la función funciona, podemos eliminarlas. El código de ese tipo se llama **andamiaje** porque ayuda a construir el programa, pero no forma parte del producto final.

Este ejemplo demuestra el desarrollo incremental. Los aspectos clave de este proceso son:

1. Empieza con un programa que funcione, haz cambios pequeños y prueba después de cada cambio.
2. Usa variables para guardar valores intermedios, de modo que puedas mostrarlos y comprobarlos.
3. Una vez que el programa funcione, elimina el andamiaje.

En cualquier punto, si hay un error, deberías tener una buena idea de dónde está. El desarrollo incremental puede ahorrarte mucho tiempo de debugging.

6.5. Funciones booleanas

Las funciones pueden devolver los valores booleanos `True` y `False`, lo que a menudo resulta conveniente para encapsular una prueba compleja en una función. Por ejemplo, `is_divisible` comprueba si `x` es divisible por `y` sin resto.

```
def is_divisible(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

Así es como la usamos.

```
is_divisible(6, 4)
```

```
is_divisible(6, 3)
```

Dentro de la función, el resultado del operador `==` es un booleano, así que podemos escribir la función de forma más concisa devolviéndolo directamente.

```
def is_divisible(x, y):
    return x % y == 0
```

Las funciones booleanas se usan a menudo en sentencias condicionales.

```
if is_divisible(6, 2):
    print('divisible')
```

Podría ser tentador escribir algo como esto:

```
if is_divisible(6, 2) == True:
    print('divisible')
```

Pero la comparación es innecesaria.

6.6. Recursión con valores de retorno

Ahora que podemos escribir funciones con valores de retorno, podemos escribir funciones recursivas con valores de retorno, y con esa capacidad hemos cruzado un umbral importante – el subconjunto de Python que tenemos ahora es **Turing completo**, lo que significa que podemos realizar cualquier cálculo que pueda describirse mediante un algoritmo.

Para demostrar la recursión con valores de retorno, evaluaremos algunas funciones matemáticas definidas de forma recursiva. Una definición recursiva se parece a una definición circular, en el sentido de que la definición se refiere a la cosa que se está definiendo. Una definición verdaderamente circular no es muy útil:

> vorpal: Un adjetivo usado para describir algo que es vorpal.

Si vieras esa definición en el diccionario, quizá te molestaría. Por otro lado, si buscaras la definición de la función factorial, denotada con el símbolo !, podrías encontrar algo como esto:

Math input error

Esta definición dice que el factorial de 0 es 1, y que el factorial de cualquier otro valor, n, es n multiplicado por el factorial de $n - 1$.

Si puedes escribir una definición recursiva de algo, puedes escribir un programa en Python para evaluarlo. Siguiendo un proceso de desarrollo incremental, empezaremos con una función que toma `n` como parámetro y siempre devuelve `0`.

```
def factorial(n):
    return 0
```

Ahora añadamos la primera parte de la definición – si resulta que el argumento es `0`, todo lo que tenemos que hacer es devolver `1`:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return 0
```

Ahora completemos la segunda parte – si `n` no es `0`, tenemos que hacer una llamada recursiva para encontrar el factorial de `n-1` y luego multiplicar el resultado por `n`:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        recurse = factorial(n-1)
        return n * recurse
```

El flujo de ejecución de este programa es similar al flujo de `countdown` en el Capítulo 5. Si llamamos a `factorial` con el valor `3`:

Como `3` no es `0`, tomamos la segunda rama y calculamos el factorial de `n-1`....

> Como `2` no es `0`, tomamos la segunda rama y calculamos el factorial de `n-1`....
>
> > Como `1` no es `0`, tomamos la segunda rama y calculamos el factorial de `n-1`....
> >
> > > Como `0` es igual a `0`, tomamos la primera rama y devolvemos `1` sin hacer más llamadas recursivas.
> >
> > El valor de retorno, `1`, se multiplica por `n`, que es `1`, y se devuelve el resultado.
>
> El valor de retorno, `1`, se multiplica por `n`, que es `2`, y se devuelve el resultado.

El valor de retorno `2` se multiplica por `n`, que es `3`, y el resultado, `6`, se convierte en el valor de retorno de la llamada a la función que inició todo el proceso.

La figura siguiente muestra el diagrama de pila para esta secuencia de llamadas a funciones.

```
from diagram import Frame, Stack, make_binding

main = Frame([], name='__main__', loc='left')
frames = [main]

ns = 3, 2, 1
recurses = 2, 1, 1
results = 6, 2, 1

for n, recurse, result in zip(ns, recurses, results):
    binding1 = make_binding('n', n)
    binding2 = make_binding('recurse', recurse)
    frame = Frame([binding1, binding2],
                  name='factorial', value=result,
                  loc='left', dx=1.2)
    frames.append(frame)

binding1 = make_binding('n', 0)
frame = Frame([binding1], name='factorial', value=1,
              shim=1.2, loc='left', dx=1.4)
frames.append(frame)

stack = Stack(frames, dy=-0.45)
```

```
from diagram import diagram, adjust

width, height, x, y = [2.74, 2.26, 0.73, 2.05]
ax = diagram(width, height)
bbox = stack.draw(ax, x, y)
# adjust(x, y, bbox)
```

Los valores de retorno se muestran pasando de vuelta hacia arriba por la pila. En cada marco, el valor de retorno es el producto de `n` y `recurse`.

En el último marco, la variable local `recurse` no existe porque la rama que la crea no se ejecuta.

6.7. Salto de fe

Seguir el flujo de ejecución es una forma de leer programas, pero puede volverse abrumadora rápidamente. Una alternativa es lo que llamo el "salto de fe". Cuando llegas a una llamada a una función, en lugar de seguir el flujo de ejecución, *asumes* que la función funciona correctamente y devuelve el resultado adecuado.

De hecho, ya estás practicando este salto de fe cuando usas funciones integradas. Cuando llamas a `abs` o `math.sqrt`, no examinas los cuerpos de esas funciones – simplemente asumes que funcionan.

Lo mismo ocurre cuando llamas a una de tus propias funciones. Por ejemplo, antes escribimos una función llamada `is_divisible` que determina si un número es divisible por otro. Una vez que nos convencemos de que esta función es correcta, podemos usarla sin volver a mirar el cuerpo.

Lo mismo ocurre con los programas recursivos. Cuando llegas a la llamada recursiva, en lugar de seguir el flujo de ejecución, deberías asumir que la llamada recursiva funciona y luego preguntarte: "Suponiendo que puedo calcular el factorial de $n - 1$, ¿puedo calcular el factorial de n?" La definición recursiva del factorial implica que sí puedes, multiplicando por n.

Por supuesto, es un poco extraño asumir que la función funciona correctamente cuando aún no has terminado de escribirla, ¡pero por eso se llama salto de fe!

6.8. Fibonacci

Después de `factorial`, el ejemplo más común de función recursiva es `fibonacci`, que tiene la siguiente definición:

Math input error

Traducida a Python, se ve así:

```
def fibonacci(n):
    if n == 0:
        return 0
    elif  n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

Si intentas seguir aquí el flujo de ejecución, incluso para valores pequeños de n, te explota la cabeza. Pero según el salto de fe, si asumes que las dos llamadas recursivas funcionan correctamente, puedes confiar en que la última sentencia `return` es correcta.

Como nota aparte, esta forma de calcular números de Fibonacci es muy ineficiente. En el Capítulo 10 explicaré por qué y sugeriré una forma de mejorarla.

6.9. Comprobación de tipos

¿Qué ocurre si llamamos a `factorial` y le damos `1.5` como argumento?

```
factorial(1.5)
```

Parece una recursión infinita. ¿Cómo puede ser? La función tiene casos base cuando `n == 1` o `n == 0`. Pero si `n` no es un entero, podemos *saltarnos* el caso base y hacer recursión para siempre.

En este ejemplo, el valor inicial de `n` es `1.5`. En la primera llamada recursiva, el valor de `n` es `0.5`. En la siguiente, es `-0.5`. A partir de ahí, se hace más pequeño (más negativo), pero nunca será `0`.

Para evitar la recursión infinita podemos usar la función integrada `isinstance` para comprobar el tipo del argumento. Así comprobamos si un valor es un entero.

```
isinstance(3, int)
```

```
isinstance(1.5, int)
```

Ahora aquí tienes una versión de `factorial` con comprobación de errores.

```
def factorial(n):
    if not isinstance(n, int):
        print('factorial is only defined for integers.')
        return None
    elif n < 0:
        print('factorial is not defined for negative numbers.')
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Primero comprueba si `n` es un entero. Si no lo es, muestra un mensaje de error y devuelve `None`.

```
factorial('crunchy frog')
```

Luego comprueba si `n` es negativo. Si lo es, muestra un mensaje de error y devuelve `None`.

```
factorial(-2)
```

Si superamos ambas comprobaciones, sabemos que `n` es un entero no negativo, así que podemos confiar en que la recursión terminará. Comprobar los parámetros de una función para asegurarse de que tienen los tipos y valores correctos se llama **validación de entrada**.

6.10. Debugging

Dividir un programa grande en funciones más pequeñas crea puntos de control naturales para depurar. Si una función no funciona, hay tres posibilidades que considerar:

- Hay algo mal con los argumentos que recibe la función – es decir, se viola una precondición.
- Hay algo mal con la función – es decir, se viola una postcondición.
- Quien llama a la función está haciendo algo mal con el valor de retorno.

Para descartar la primera posibilidad, puedes añadir una sentencia `print` al principio de la función que muestre los valores de los parámetros (y quizá sus tipos). O puedes escribir código que compruebe explícitamente las precondiciones.

Si los parámetros parecen correctos, puedes añadir una sentencia `print` antes de cada sentencia `return` y mostrar el valor de retorno. Si es posible, llama a la función con argumentos que faciliten comprobar el resultado.

Si la función parece funcionar, mira la llamada a la función para asegurarte de que el valor de retorno se está usando correctamente – ¡o de que se está usando siquiera!

Añadir sentencias `print` al principio y al final de una función puede ayudar a hacer más visible el flujo de ejecución. Por ejemplo, aquí tienes una versión de `factorial` con sentencias print:

```
def factorial(n):
    space = ' ' * (4 * n)
    print(space, 'factorial', n)
    if n == 0:
        print(space, 'returning 1')
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        print(space, 'returning', result)
        return result
```

`space` es una cadena de caracteres de espacio que controla la indentación de la salida. Este es el resultado de `factorial(3)` :

```
factorial(3)
```

Si tienes dudas sobre el flujo de ejecución, este tipo de salida puede ser útil. Lleva tiempo desarrollar andamiaje eficaz, pero un poco de andamiaje puede ahorrar mucho debugging.

6.11. Glosario

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

función pura: Una función que no muestra nada ni tiene ningún otro efecto, aparte de devolver un valor de retorno.

código muerto: Parte de un programa que nunca puede ejecutarse, a menudo porque aparece después de una sentencia `return`.

desarrollo incremental: Un plan de desarrollo de programas pensado para evitar la depuración añadiendo y probando solo una pequeña cantidad de código cada vez.

andamiaje: Código que se usa durante el desarrollo del programa pero que no forma parte de la versión final.

Turing completo: Un lenguaje, o un subconjunto de un lenguaje, es Turing completo si puede realizar cualquier cálculo que pueda describirse mediante un algoritmo.

validación de entrada: Comprobar los parámetros de una función para asegurarse de que tienen los tipos y valores correctos

6.12. Ejercicios

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

6.12.1. Pregunta a un asistente virtual

En este capítulo vimos una función incorrecta que puede terminar sin devolver un valor.

```
def absolute_value_wrong(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```

Y una versión de la misma función que tiene código muerto al final.

```
def absolute_value_extra_return(x):
    if x < 0:
        return -x
    else:
        return x

    return 'This is dead code.'
```

Y vimos el siguiente ejemplo, que es correcto pero no idiomático.

```
def is_divisible(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

Pregunta a un asistente virtual qué está mal en cada una de estas funciones y comprueba si puede detectar los errores o mejorar el estilo.

Luego pregunta: "Escribe una función que tome las coordenadas de dos puntos y calcule la distancia entre ellos." Comprueba si el resultado se parece a la versión de `distance` que escribimos en este capítulo.

6.12.2. Ejercicio

Usa desarrollo incremental para escribir una función llamada `hypot` que devuelva la longitud de la hipotenusa de un triángulo rectángulo dadas las longitudes de los otros dos catetos como argumentos.

Nota: Hay una función en el módulo math llamada `hypot` que hace lo mismo, ¡pero no deberías usarla para este ejercicio!

Aunque puedas escribir la función correctamente al primer intento, empieza con una función que siempre devuelva `0` y practica haciendo cambios pequeños, probando a medida que avanzas. Cuando termines, la función solo debería devolver un valor – no debería mostrar nada.

6.12.3. Ejercicio

Escribe una función booleana, `is_between(x, y, z)`, que devuelva `True` si $x < y < z$ o si $z < y < x$, y `False` en caso contrario.

Puedes usar estos ejemplos para probar tu función.

```
is_between(1, 2, 3)  # should be True
```

```
is_between(3, 2, 1)  # should be True
```

```
is_between(1, 3, 2)  # should be False
```

```
is_between(2, 3, 1)  # should be False
```

6.12.4. Ejercicio

La función de Ackermann, $A(m, n)$, se define así:

Math input error

Escribe una función llamada `ackermann` que evalúe la función de Ackermann. ¿Qué ocurre si llamas a `ackermann(5, 5)`?

Puedes usar estos ejemplos para probar tu función.

```
ackermann(3, 2)  # should be 29
```

```
ackermann(3, 3)  # should be 61
```

```
ackermann(3, 4)  # should be 125
```

Si llamas a esta función con valores mayores que 4, obtienes un `RecursionError`.

```
ackermann(5, 5)
```

Para ver por qué, añade una sentencia print al principio de la función para mostrar los valores de los parámetros, y luego ejecuta los ejemplos de nuevo.

6.12.5. Ejercicio

Un número, a, es una potencia de b si es divisible por b y a/b es una potencia de b. Escribe una función llamada `is_power` que tome los parámetros `a` y `b`, y devuelva `True` si `a` es una potencia de `b`. Nota: tendrás que pensar en el caso base.

Puedes usar estos ejemplos para probar tu función.

```
is_power(65536, 2)   # should be True
```

```
is_power(27, 3)  # should be True
```

```
is_power(24, 2)  # should be False
```

```
is_power(1, 17)   # should be True
```

6.12.6. Ejercicio

El máximo común divisor (MCD) de a y b es el número más grande que divide a ambos sin dejar resto.

Una forma de encontrar el MCD de dos números se basa en la observación de que si r es el resto cuando a se divide por b, entonces $gcd(a, b) = gcd(b, r)$. Como caso base, podemos usar $gcd(a, 0) = a$.

Escribe una función llamada `gcd` que tome los parámetros `a` y `b`, y devuelva su máximo común divisor.

Puedes usar estos ejemplos para probar tu función.

```
gcd(12, 8)    # should be 4
```

```
gcd(13, 17)   # should be 1
```

Think Python: 3rd Edition

Traducción al español por midudev (Miguel Ángel Durán).

Puedes comprar versiones impresas y libro electrónico de *Think Python 3e* en Bookshop.org y Amazon.

7. Iteración y búsqueda

En 1939, Ernest Vincent Wright publicó una novela de 50.000 palabras llamada *Gadsby* que no contiene la letra "e". Como "e" es la letra más común en inglés, escribir incluso unas pocas palabras sin usarla es difícil. Para hacernos una idea de lo difícil que es, en este capítulo calcularemos la fracción de palabras en inglés que tienen al menos una "e".

Para ello, usaremos sentencias `for` para recorrer las letras de una cadena y las palabras de un archivo, y actualizaremos variables en un bucle para contar el número de palabras que contienen una "e". Usaremos el operador `in` para comprobar si una letra aparece en una palabra, y aprenderás un patrón de programación llamado "búsqueda lineal".

Como ejercicio, usarás estas herramientas para resolver un juego de palabras llamado “Spelling Bee”.

7.1. Bucles y cadenas

En el capítulo 3 vimos un bucle `for` que usa la función `range` para mostrar una secuencia de números.

```
for i in range(3):
    print(i, end=' ')
```

Esta versión usa el argumento de palabra clave `end` para que la función `print` añada un espacio después de cada número en lugar de un salto de línea.

También podemos usar un bucle `for` para mostrar las letras de una cadena.

```
for letter in 'Gadsby':
    print(letter, end=' ')
```

Fíjate en que he cambiado el nombre de la variable de `i` a `letter`, lo que proporciona más información sobre el valor al que se refiere. La variable definida en un bucle `for` se llama **variable de bucle**.

Ahora que podemos recorrer las letras de una palabra, podemos comprobar si contiene la letra “e”.

```
for letter in "Gadsby":
    if letter == 'E' or letter == 'e':
        print('This word has an "e"')
```

Antes de seguir, encapsulemos ese bucle en una función.

```
def has_e():
    for letter in "Gadsby":
        if letter == 'E' or letter == 'e':
            print('This word has an "e"')
```

Y convirtámosla en una función pura que devuelva `True` si la palabra contiene una "e" y `False` en caso contrario.

```
def has_e():
    for letter in "Gadsby":
        if letter == 'E' or letter == 'e':
            return True
    return False
```

Podemos generalizarla para que tome la palabra como parámetro.

```
def has_e(word):
    for letter in word:
        if letter == 'E' or letter == 'e':
            return True
    return False
```

Ahora podemos probarla así:

```
has_e('Gadsby')
```

```
has_e('Emma')
```

7.2. Leer la lista de palabras

Para ver cuántas palabras contienen una "e", necesitaremos una lista de palabras. La que usaremos es una lista de unas 114.000 palabras oficiales de crucigramas; es decir, palabras que se consideran válidas en crucigramas y otros juegos de palabras.

La siguiente celda descarga la lista de palabras, que es una versión modificada de una lista recopilada y aportada al dominio público por Grady Ward como parte del proyecto léxico Moby (consulta http://wikipedia.org/wiki/Moby_Project).

```
download('https://raw.githubusercontent.com/AllenDowney/ThinkPython/v3/words.txt');
```

La lista de palabras está en un archivo llamado `words.txt`, que se descarga en el notebook de este capítulo. Para leerlo, usaremos la función integrada `open`, que toma el nombre del archivo como parámetro y devuelve un **objeto de archivo** que podemos usar para leer el archivo.

```
file_object = open('words.txt')
```

El objeto de archivo proporciona un método llamado `readline`, que lee caracteres del archivo hasta llegar a un salto de línea y devuelve el resultado como una cadena:

```
file_object.readline()
```

Fíjate en que la sintaxis para llamar a `readline` es diferente de la de las funciones que hemos visto hasta ahora. Eso se debe a que es un **método**, que es una función asociada a un objeto. En este caso `readline` está asociado al objeto de archivo, así que lo llamamos usando el nombre del objeto, el operador punto y el nombre del método.

La primera palabra de la lista es "aa", que es un tipo de lava. La secuencia `\n` representa el carácter de salto de línea que separa esta palabra de la siguiente.

El objeto de archivo lleva la cuenta de en qué parte del archivo está, así que si llamas a `readline` de nuevo, obtienes la siguiente palabra:

```
line = file_object.readline()
line
```

Para eliminar el salto de línea del final de la palabra, podemos usar `strip`, que es un método asociado con las cadenas, así que podemos llamarlo así.

```
word = line.strip()
word
```

`strip` elimina caracteres de espacio en blanco – incluidos espacios, tabulaciones y saltos de línea – del principio y el final de la cadena.

También puedes usar un objeto de archivo como parte de un bucle `for`. Este programa lee `words.txt` e imprime cada palabra, una por línea:

```
for line in open('words.txt'):
    word = line.strip()
    print(word)
```

Ahora que podemos leer la lista de palabras, el siguiente paso es contarlas. Para eso, necesitaremos la capacidad de actualizar variables.

7.3. Actualizar variables

Como quizá hayas descubierto, es legal hacer más de una asignación a la misma variable. Una nueva asignación hace que una variable existente se refiera a un valor nuevo (y deje de referirse al valor anterior).

Por ejemplo, aquí hay una asignación inicial que crea una variable.

```
x = 5
x
```

Y aquí hay una asignación que cambia el valor de una variable.

```
x = 7
x
```

La siguiente figura muestra cómo se ven estas asignaciones en un diagrama de estado.

```
from diagram import make_rebind, draw_bindings

bindings = make_rebind('x', [5, 7])
```

```
from diagram import diagram, adjust

width, height, x, y = [0.54, 0.61, 0.07, 0.45]
ax = diagram(width, height)
bbox = draw_bindings(bindings, ax, x, y)
# adjust(x, y, bbox)
```

La flecha punteada indica que `x` ya no se refiere a `5`. La flecha sólida indica que ahora se refiere a `7`.

Un tipo común de asignación es una **actualización**, donde el nuevo valor de la variable depende del anterior.

```
x = 7
```

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

Esta sentencia significa "obtén el valor actual de `x`, suma uno y asigna el resultado de nuevo a `x`."

Si intentas actualizar una variable que no existe, obtienes un error, porque Python evalúa la expresión de la derecha antes de asignar un valor a la variable de la izquierda.

```
z = z + 1
```

Antes de poder actualizar una variable, tienes que **inicializarla**, normalmente con una asignación sencilla:

```
z = 0
z = z + 1
z
```

Aumentar el valor de una variable se llama **incremento**; disminuir el valor se llama **decremento**. Como estas operaciones son tan comunes, Python proporciona **operadores de asignación aumentada** que actualizan una variable de forma más concisa. Por ejemplo, el operador `+=` incrementa una variable en la cantidad dada.

```
z += 2
z
```

Hay operadores de asignación aumentada para los otros operadores aritméticos, incluidos `-=` y `*=`.

7.4. Recorrer y contar

El siguiente programa cuenta el número de palabras en la lista de palabras.

```
total = 0

for line in open('words.txt'):
    word = line.strip()
    total += 1
```

Empieza inicializando `total` a `0`. Cada vez que pasa por el bucle, incrementa `total` en `1`. Así que cuando el bucle termina, `total` se refiere al número total de palabras.

```
total
```

Una variable como esta, usada para contar el número de veces que ocurre algo, se llama **contador**.

Podemos añadir un segundo contador al programa para llevar la cuenta del número de palabras que contienen una "e".

```
total = 0
count = 0

for line in open('words.txt'):
    word = line.strip()
    total = total + 1
    if has_e(word):
        count += 1
```

Veamos cuántas palabras contienen una "e".

```
count
```

Como porcentaje de `total`, alrededor de dos tercios de las palabras usan la letra "e".

```
count / total * 100
```

Así puedes entender por qué es difícil escribir un libro sin usar ninguna de esas palabras.

7.5. El operador in

La versión de `has_e` que escribimos en este capítulo es más complicada de lo necesario. Python proporciona un operador, `in`, que comprueba si un carácter aparece en una cadena.

```
word = 'Gadsby'
'e' in word
```

Así que podemos reescribir `has_e` así.

```
def has_e(word):
    if 'E' in word or 'e' in word:
        return True
    else:
        return False
```

Y como la condición de la sentencia `if` tiene un valor booleano, podemos eliminar la sentencia `if` y devolver el valor booleano directamente.

```
def has_e(word):
    return 'E' in word or 'e' in word
```

Podemos simplificar esta función todavía más usando el método `lower`, que convierte las letras de una cadena a minúsculas. Aquí tienes un ejemplo.

```
word.lower()
```

`lower` crea una cadena nueva – no modifica la cadena existente – así que el valor de `word` no cambia.

```
word
```

Así es como podemos usar `lower` en `has_e`.

```
def has_e(word):
    return 'e' in word.lower()
```

```
has_e('Gadsby')
```

```
has_e('Emma')
```

7.6. Búsqueda

Basándonos en esta versión más sencilla de `has_e`, escribamos una función más general llamada `uses_any` que recibe un segundo parámetro que es una cadena de letras. Devuelve `True` si la palabra usa cualquiera de las letras y `False` en caso contrario.

```
def uses_any(word, letters):
    for letter in word.lower():
        if letter in letters.lower():
            return True
    return False
```

Aquí tienes un ejemplo donde el resultado es `True`.

```
uses_any('banana', 'aeiou')
```

Y otro donde es `False`.

```
uses_any('apple', 'xyz')
```

`uses_any` convierte `word` y `letters` a minúsculas, así que funciona con cualquier combinación de mayúsculas y minúsculas.

```
uses_any('Banana', 'AEIOU')
```

La estructura de `uses_any` es similar a la de `has_e`. Recorre las letras de `word` y las comprueba una por una. Si encuentra una que aparece en `letters`, devuelve `True` inmediatamente. Si llega hasta el final del bucle sin encontrar ninguna, devuelve `False`.

Este patrón se llama **búsqueda lineal**. En los ejercicios al final de este capítulo, escribirás más funciones que usan este patrón.

7.7. Doctest

En el capítulo 4 usamos un docstring para documentar una función – es decir, para explicar qué hace. También es posible usar un docstring para *probar* una función. Aquí tienes una versión de `uses_any` con un docstring que incluye pruebas.

```
def uses_any(word, letters):
    """Checks if a word uses any of a list of letters.

    >>> uses_any('banana', 'aeiou')
    True
    >>> uses_any('apple', 'xyz')
    False
    """
    for letter in word.lower():
        if letter in letters.lower():
            return True
    return False
```

Cada prueba empieza con `>>>`, que se usa como indicador en algunos entornos de Python para indicar dónde el usuario puede escribir código. En un doctest, al indicador le sigue una expresión, normalmente una llamada a función. La línea siguiente indica el valor que debería tener la expresión si la función funciona correctamente.

En el primer ejemplo, `'banana'` usa `'a'`, así que el resultado debería ser `True`. En el segundo ejemplo, `'apple'` no usa ninguna de `'xyz'`, así que el resultado debería ser `False`.

Para ejecutar estas pruebas, tenemos que importar el módulo `doctest` y ejecutar una función llamada `run_docstring_examples`. Para que esta función sea más fácil de usar, escribí la siguiente función, que toma un objeto que representa una función como argumento.

```
from doctest import run_docstring_examples

def run_doctests(func):
    run_docstring_examples(func, globals(), name=func.__name__)
```

Todavía no hemos aprendido sobre `globals` y `__name__` – puedes ignorarlos. Ahora podemos probar `uses_any` así.

```
run_doctests(uses_any)
```

`run_doctests` encuentra las expresiones en el docstring y las evalúa. Si el resultado es el valor esperado, la prueba **pasa**. En caso contrario, **falla**. Si todas las pruebas pasan, `run_doctests` no muestra ninguna salida – en ese caso, no tener noticias es buena señal. Para ver qué ocurre cuando una prueba falla, aquí tienes una versión incorrecta de `uses_any`.

```
def uses_any_incorrect(word, letters):
    """Checks if a word uses any of a list of letters.

    >>> uses_any_incorrect('banana', 'aeiou')
    True
    >>> uses_any_incorrect('apple', 'xyz')
    False
    """
    for letter in word.lower():
        if letter in letters.lower():
            return True
        else:
            return False     # INCORRECT!
```

Y esto es lo que ocurre cuando la probamos.

```
run_doctests(uses_any_incorrect)
```

La salida incluye el ejemplo que falló, el valor que se esperaba que devolviera la función y el valor que produjo realmente.

Si no tienes claro por qué falló esta prueba, tendrás la oportunidad de depurarla como ejercicio.

7.8. Glosario

variable de bucle: Una variable definida en la cabecera de un bucle `for`.

objeto de archivo: Un objeto que representa un archivo abierto y lleva la cuenta de qué partes del archivo se han leído o escrito.

método: Una función asociada a un objeto que se llama usando el operador punto.

actualización: Una sentencia de asignación que da un valor nuevo a una variable que ya existe, en lugar de crear una variable nueva.

inicializar: Crear una variable nueva y darle un valor.

incremento: Aumentar el valor de una variable.

decremento: Disminuir el valor de una variable.

contador: Una variable usada para contar algo, normalmente inicializada a cero y luego incrementada.

búsqueda lineal: Un patrón computacional que busca en una secuencia de elementos y se detiene cuando encuentra lo que está buscando.

pasar: Si una prueba se ejecuta y el resultado es el esperado, la prueba pasa.

fallar: Si una prueba se ejecuta y el resultado no es el esperado, la prueba falla.

7.9. Ejercicios

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

7.9.1. Pregunta a un asistente virtual

En `uses_any`, quizá hayas notado que la primera sentencia `return` está dentro del bucle y la segunda está fuera.

```
def uses_any(word, letters):
    for letter in word.lower():
        if letter in letters.lower():
            return True
    return False
```

Cuando la gente escribe por primera vez funciones como esta, es un error común poner ambas sentencias `return` dentro del bucle, así.

```
def uses_any_incorrect(word, letters):
    for letter in word.lower():
        if letter in letters.lower():
            return True
        else:
            return False     # INCORRECT!
```

Pregunta a un asistente virtual qué está mal en esta versión.

7.9.2. Ejercicio

Escribe una función llamada `uses_none` que reciba una palabra y una cadena de letras prohibidas, y devuelva `True` si la palabra no usa ninguna de las letras prohibidas.

Aquí tienes un esquema de la función que incluye dos doctests. Completa la función para que pase estas pruebas, y añade al menos un doctest más.

```
def uses_none(word, forbidden):
    """Checks whether a word avoid forbidden letters.

    >>> uses_none('banana', 'xyz')
    True
    >>> uses_none('apple', 'efg')
    False
    """
    return None
```

```
run_doctests(uses_none)
```

7.9.3. Ejercicio

Escribe una función llamada `uses_only` que reciba una palabra y una cadena de letras, y que devuelva `True` si la palabra contiene solo letras de la cadena.

Aquí tienes un esquema de la función que incluye dos doctests. Completa la función para que pase estas pruebas, y añade al menos un doctest más.

```
def uses_only(word, available):
    """Checks whether a word uses only the available letters.

    >>> uses_only('banana', 'ban')
    True
    >>> uses_only('apple', 'apl')
    False
    """
    return None
```

```
run_doctests(uses_only)
```

7.9.4. Ejercicio

Escribe una función llamada `uses_all` que reciba una palabra y una cadena de letras, y que devuelva `True` si la palabra contiene todas las letras de la cadena al menos una vez.

Aquí tienes un esquema de la función que incluye dos doctests. Completa la función para que pase estas pruebas, y añade al menos un doctest más.

```
def uses_all(word, required):
    """Checks whether a word uses all required letters.

    >>> uses_all('banana', 'ban')
    True
    >>> uses_all('apple', 'api')
    False
    """
    return None
```

```
run_doctests(uses_all)
```

7.9.5. Ejercicio

The New York Times publica un juego diario llamado “Spelling Bee” que reta a los lectores a formar tantas palabras como sea posible usando solo siete letras, donde una de las letras es obligatoria. Las palabras deben tener al menos cuatro letras.

Por ejemplo, el día que escribí esto, las letras eran `ACDLORT`, con `R` como letra obligatoria. Así que "color" es una palabra aceptable, pero "told" no, porque no usa `R`, y "rat" tampoco porque solo tiene tres letras. Las letras se pueden repetir, así que "ratatat" es aceptable.

Escribe una función llamada `check_word` que compruebe si una palabra dada es aceptable. Debe recibir como parámetros la palabra que se va a comprobar, una cadena de siete letras disponibles y una cadena que contiene la única letra obligatoria. Puedes usar las funciones que escribiste en ejercicios anteriores.

Aquí tienes un esquema de la función que incluye doctests. Completa la función y luego comprueba que todas las pruebas pasan.

```
def check_word(word, available, required):
    """Check whether a word is acceptable.

    >>> check_word('color', 'ACDLORT', 'R')
    True
    >>> check_word('ratatat', 'ACDLORT', 'R')
    True
    >>> check_word('rat', 'ACDLORT', 'R')
    False
    >>> check_word('told', 'ACDLORT', 'R')
    False
    >>> check_word('bee', 'ACDLORT', 'R')
    False
    """
    return False
```

```
run_doctests(check_word)
```

Según las reglas de "Spelling Bee",

- Las palabras de cuatro letras valen 1 punto cada una.
- Las palabras más largas obtienen 1 punto por letra.
- Cada juego incluye al menos un "pangram" que usa todas las letras. ¡Estos valen 7 puntos extra!

Escribe una función llamada `score_word` que reciba una palabra y una cadena de letras disponibles y devuelva su puntuación. Puedes asumir que la palabra es aceptable.

De nuevo, aquí tienes un esquema de la función con doctests.

```
def word_score(word, available):
    """Compute the score for an acceptable word.

    >>> word_score('card', 'ACDLORT')
    1
    >>> word_score('color', 'ACDLORT')
    5
    >>> word_score('cartload', 'ACDLORT')
    15
    """
    return 0
```

```
run_doctests(word_score)
```

Cuando todas tus funciones pasen sus pruebas, usa el siguiente bucle para buscar palabras aceptables en la lista de palabras y sumar sus puntuaciones.

```
available = 'ACDLORT'
required = 'R'

total = 0

file_object = open('words.txt')
for line in file_object:
    word = line.strip()
    if check_word(word, available, required):
        score = word_score(word, available)
        total = total + score
        print(word, score)

print("Total score", total)
```

Visita la página de “Spelling Bee” en https://www.nytimes.com/puzzles/spelling-bee y escribe las letras disponibles del día. La letra del centro es obligatoria.

Encontré un conjunto de letras que forma palabras con una puntuación total de 5820. ¿Puedes superarlo? Encontrar el mejor conjunto de letras podría ser demasiado difícil – hay que ser realista.

7.9.6. Ejercicio

Quizá hayas notado que las funciones que escribiste en los ejercicios anteriores tenían mucho en común. De hecho, son tan parecidas que a menudo puedes usar una función para escribir otra.

Por ejemplo, si una palabra no usa ninguna letra de un conjunto de letras prohibidas, eso significa que no usa ninguna. Así que podemos escribir una versión de `uses_none` así.

```
def uses_none(word, forbidden):
    """Checks whether a word avoids forbidden letters.

    >>> uses_none('banana', 'xyz')
    True
    >>> uses_none('apple', 'efg')
    False
    >>> uses_none('', 'abc')
    True
    """
    return not uses_any(word, forbidden)
```

```
run_doctests(uses_none)
```

También hay una similitud entre `uses_only` y `uses_all` que puedes aprovechar. Si tienes una versión funcional de `uses_only`, mira si puedes escribir una versión de `uses_all` que llame a `uses_only`.

7.9.7. Ejercicio

Si te atascaste en la pregunta anterior, prueba a preguntar a un asistente virtual: "Dada una función, `uses_only`, que recibe dos cadenas y comprueba que la primera usa solo las letras del segundo, úsala para escribir `uses_all`, que recibe dos cadenas y comprueba si la primera usa todas las letras del segundo, permitiendo repeticiones."

Usa `run_doctests` para comprobar la respuesta.

```
run_doctests(uses_all)
```

7.9.8. Ejercicio

Ahora veamos si podemos escribir `uses_all` basándonos en `uses_any`.

Pregunta a un asistente virtual: "Dada una función, `uses_any`, que recibe dos cadenas y comprueba si la primera usa cualquiera de las letras del segundo, ¿puedes usarla para escribir

`uses_all`, que recibe dos cadenas y comprueba si la primera usa todas las letras del segundo, permitiendo repeticiones?"

Si dice que puede, ¡asegúrate de probar el resultado!

```
# Here's what I got from ChatGPT 4o December 26, 2024
# It's correct, but it makes multiple calls to uses_any

def uses_all(s1, s2):
    """Checks if all characters in s2 are in s1, allowing repeats."""
    for char in s2:
        if not uses_any(s1, char):
            return False
    return True
```


8. Cadenas y expresiones regulares

Las cadenas no son como los enteros, los flotantes y los booleanos. Una cadena es una **secuencia**, lo que significa que contiene múltiples valores en un orden particular. En este capítulo veremos cómo acceder a los valores que componen una cadena, y usaremos funciones que procesan cadenas.

También usaremos expresiones regulares, que son una herramienta poderosa para encontrar patrones en una cadena y realizar operaciones como búsqueda y reemplazo.

Como ejercicio, tendrás la oportunidad de aplicar estas herramientas a un juego de palabras llamado Wordle.

8.1. Una cadena es una secuencia

Una cadena es una secuencia de caracteres. Un **carácter** puede ser una letra (en casi cualquier alfabeto), un dígito, un signo de puntuación o un espacio en blanco.

Puedes seleccionar un carácter de una cadena con el operador de corchetes. Esta sentencia de ejemplo selecciona el carácter número 1 de `fruit` y lo asigna a `letter`:

```
fruit = 'banana'
letter = fruit[1]
```

La expresión entre corchetes es un **índice**, llamado así porque *indica* qué carácter de la secuencia seleccionar. Pero el resultado quizá no sea lo que esperas.

```
letter
```

La letra con índice `1` es en realidad la segunda letra de la cadena. Un índice es un desplazamiento desde el principio de la cadena, así que el desplazamiento de la primera letra es `0`.

```
fruit[0]
```

Puedes pensar en `'b'` como la letra 0 de `'banana'` – pronunciada “ceroésima”.

El índice entre corchetes puede ser una variable.

```
i = 1
fruit[i]
```

O una expresión que contiene variables y operadores.

```
fruit[i+1]
```

Pero el valor del índice tiene que ser un entero – de lo contrario obtienes un `TypeError`.

```
fruit[1.5]
```

Como vimos en el capítulo 1, podemos usar la función integrada `len` para obtener la longitud de una cadena.

```
n = len(fruit)
n
```

Para obtener la última letra de una cadena, podrías sentir la tentación de escribir esto:

```
fruit[n]
```

Pero eso causa un `IndexError` porque no hay ninguna letra en `'banana'` con el índice 6. Como empezamos a contar en `0`, las seis letras están numeradas de `0` a `5`. Para obtener el último carácter, tienes que restar `1` a `n`:

```
fruit[n-1]
```

Pero hay una forma más sencilla. Para obtener la última letra de una cadena, puedes usar un índice negativo, que cuenta hacia atrás desde el final.

```
fruit[-1]
```

El índice `-1` selecciona la última letra, `-2` selecciona la penúltima, y así sucesivamente.

8.2. Segmentos de cadenas

Un segmento de una cadena se llama **segmento**. Seleccionar un segmento es similar a seleccionar un carácter.

```
fruit = 'banana'
fruit[0:3]
```

El operador `[n:m]` devuelve la parte de la cadena desde el carácter `n`-ésimo hasta el carácter `m`-ésimo, incluyendo el primero pero excluyendo el segundo. Este comportamiento es contraintuitivo, pero puede ayudar imaginar que los índices apuntan *entre* los caracteres, como en esta figura:

```
from diagram import make_binding, Element, Value

binding = make_binding("fruit", ' b a n a n a ')
elements = [Element(Value(i), None) for i in range(7)]
```

```
import matplotlib.pyplot as plt
from diagram import diagram, adjust
from matplotlib.transforms import Bbox

width, height, x, y = [1.35, 0.54, 0.23, 0.39]

ax = diagram(width, height)
bbox = binding.draw(ax, x, y)
bboxes = [bbox]

def draw_elts(x, y, elements):
    for elt in elements:
        bbox = elt.draw(ax, x, y, draw_value=False)
        bboxes.append(bbox)

        x1 = (bbox.xmin + bbox.xmax) / 2
        y1 = bbox.ymax + 0.02
        y2 = y1 + 0.14
        handle = plt.plot([x1, x1], [y1, y2], ':', lw=0.5, color='gray')
        x += 0.105

draw_elts(x + 0.48, y - 0.25, elements)
bbox = Bbox.union(bboxes)
# adjust(x, y, bbox)
```

Por ejemplo, el segmento `[3:6]` selecciona las letras `ana`, lo que significa que `6` es legal como parte de un segmento, pero no como índice.

Si omites el primer índice, el segmento empieza al principio de la cadena.

```
fruit[:3]
```

Si omites el segundo índice, el segmento va hasta el final de la cadena:

```
fruit[3:]
```

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

```
fruit[3:3]
```

Una cadena vacío no contiene caracteres y tiene longitud 0.

Continuando con este ejemplo, ¿qué crees que significa `fruit[:]`? Pruébalo y mira.

```
fruit[:]
```

8.3. Las cadenas son inmutables

Es tentador usar el operador `[]` en el lado izquierdo de una asignación, con la intención de cambiar un carácter en una cadena, así:

```
greeting = 'Hello, world!'
greeting[0] = 'J'
```

El resultado es un `TypeError`. En el mensaje de error, el "objeto" es la cadena y el "item" es el carácter que intentamos asignar. Por ahora, un **objeto** es lo mismo que un valor, pero refinaremos esa definición más adelante.

La razón de este error es que las cadenas son **inmutables**, lo que significa que no puedes cambiar una cadena existente. Lo máximo que puedes hacer es crear una cadena nueva que sea una variación del original.

```
new_greeting = 'J' + greeting[1:]
new_greeting
```

Este ejemplo concatena una nueva primera letra con un segmento de `greeting`. No tiene ningún efecto sobre la cadena original.

```
greeting
```

8.4. Comparación de cadenas

Los operadores relacionales funcionan con cadenas. Para ver si dos cadenas son iguales, podemos usar el operador `==`.

```
word = 'banana'

if word == 'banana':
    print('All right, banana.')
```

Otras operaciones relacionales son útiles para poner palabras en orden alfabético:

```
def compare_word(word):
    if word < 'banana':
        print(word, 'comes before banana.')
    elif word > 'banana':
        print(word, 'comes after banana.')
    else:
        print('All right, banana.')
```

```
compare_word('apple')
```

Python no trata las letras mayúsculas y minúsculas de la misma manera que lo hacen las personas. Todas las letras mayúsculas van antes que todas las minúsculas, así que:

```
compare_word('Pineapple')
```

Para resolver este problema, podemos convertir las cadenas a un formato estándar, como todo en minúsculas, antes de realizar la comparación. Tenlo en cuenta si tienes que defenderte de un hombre armado con una piña.

8.5. Métodos de cadenas

Las cadenas proporcionan métodos que realizan una variedad de operaciones útiles. Un método es similar a una función – recibe argumentos y devuelve un valor – pero la sintaxis es diferente. Por ejemplo, el método `upper` recibe una cadena y devuelve una cadena nueva con todas las letras en mayúsculas.

En lugar de la sintaxis de función `upper(word)`, usa la sintaxis de método `word.upper()`.

```
word = 'banana'
new_word = word.upper()
new_word
```

Este uso del operador punto especifica el nombre del método, `upper`, y el nombre de la cadena al que aplicar el método, `word`. Los paréntesis vacíos indican que este método no toma argumentos.

Una llamada a método se llama **invocación**; en este caso, diríamos que estamos invocando `upper` sobre `word`.

8.6. Escribir archivos

Los operadores y métodos de cadena son útiles para leer y escribir archivos de texto. Como ejemplo, trabajaremos con el texto de *Dracula*, una novela de Bram Stoker que está disponible en Project Gutenberg (https://www.gutenberg.org/ebooks/345).

```
import os

if not os.path.exists('pg345.txt'):
    !wget https://www.gutenberg.org/cache/epub/345/pg345.txt
```

He descargado el libro en un archivo de texto plano llamado `pg345.txt`, que podemos abrir para lectura así:

```
reader = open('pg345.txt')
```

Además del texto del libro, este archivo contiene una sección al principio con información sobre el libro y una sección al final con información sobre la licencia. Antes de procesar el texto, podemos eliminar este material extra encontrando las líneas especiales al principio y al final que empiezan con `'***'`.

La siguiente función recibe una línea y comprueba si es una de las líneas especiales. Usa el método `startswith`, que comprueba si una cadena empieza con una secuencia determinada de caracteres.

```
def is_special_line(line):
    return line.startswith('*** ')
```

Podemos usar esta función para recorrer las líneas del archivo e imprimir solo las líneas especiales.

```
for line in reader:
    if is_special_line(line):
        print(line.strip())
```

Ahora creemos un archivo nuevo, llamado `pg345_cleaned.txt`, que contenga solo el texto del libro. Para recorrer el libro de nuevo, tenemos que abrirlo otra vez para lectura. Y, para escribir un archivo nuevo, podemos abrirlo para escritura.

```
reader = open('pg345.txt')
writer = open('pg345_cleaned.txt', 'w')
```

`open` recibe un parámetro opcional que especifica el "modo" – en este ejemplo, `'w'` indica que estamos abriendo el archivo para escritura. Si el archivo no existe, se creará; si ya existe, el contenido será reemplazado.

Como primer paso, recorreremos el archivo hasta encontrar la primera línea especial.

```
for line in reader:
    if is_special_line(line):
        break
```

La sentencia `break` "rompe" el bucle – es decir, hace que el bucle termine inmediatamente, antes de llegar al final del archivo.

Cuando el bucle termina, `line` contiene la línea especial que hizo que la condición fuera verdadera.

```
line
```

Como `reader` lleva la cuenta de en qué parte del archivo está, podemos usar un segundo bucle para continuar donde lo dejamos.

El siguiente bucle lee el resto del archivo, una línea cada vez. Cuando encuentra la línea especial que indica el final del texto, rompe el bucle. En caso contrario, escribe la línea en el archivo de salida.

```
for line in reader:
    if is_special_line(line):
        break
    writer.write(line)
```

Cuando este bucle termina, `line` contiene la segunda línea especial.

```
line
```

En este punto `reader` y `writer` siguen abiertos, lo que significa que podríamos seguir leyendo líneas de `reader` o escribiendo líneas en `writer`. Para indicar que hemos terminado, podemos cerrar ambos archivos invocando el método `close`.

```
reader.close()
writer.close()
```

Para comprobar si este proceso tuvo éxito, podemos leer las primeras líneas del archivo nuevo que acabamos de crear.

```
for line in open('pg345_cleaned.txt'):
    line = line.strip()
    if len(line) > 0:
        print(line)
    if line.endswith('Stoker'):
        break
```

El método `endswith` comprueba si una cadena termina con una secuencia determinada de caracteres.

8.7. Buscar y reemplazar

En la traducción islandesa de *Dracula* de 1901, el nombre de uno de los personajes se cambió de “Jonathan” a “Thomas”. Para hacer este cambio en la versión inglesa, podemos recorrer el libro,

usar el método `replace` para reemplazar un nombre por otro y escribir el resultado en un archivo nuevo.

Empezaremos contando las líneas en la versión limpia del archivo.

```
total = 0
for line in open('pg345_cleaned.txt'):
    total += 1

total
```

Para ver si una línea contiene "Jonathan", podemos usar el operador `in`, que comprueba si esta secuencia de caracteres aparece en cualquier parte de la línea.

```
total = 0
for line in open('pg345_cleaned.txt'):
    if 'Jonathan' in line:
        total += 1

total
```

Hay 199 líneas que contienen el nombre, pero ese no es exactamente el número total de veces que aparece, porque puede aparecer más de una vez en una línea. Para obtener el total, podemos usar el método `count`, que devuelve el número de veces que una secuencia aparece en una cadena.

```
total = 0
for line in open('pg345_cleaned.txt'):
    total += line.count('Jonathan')

total
```

Ahora podemos reemplazar `'Jonathan'` por `'Thomas'` así:

```
writer = open('pg345_replaced.txt', 'w')

for line in open('pg345_cleaned.txt'):
    line = line.replace('Jonathan', 'Thomas')
    writer.write(line)
```

El resultado es un archivo nuevo llamado `pg345_replaced.txt` que contiene una versión de *Dracula* donde Jonathan Harker se llama Thomas.

```
total = 0
for line in open('pg345_replaced.txt'):
    total += line.count('Thomas')

total
```

8.8. Expresiones regulares

Si sabemos exactamente qué secuencia de caracteres estamos buscando, podemos usar el operador `in` para encontrarla y el método `replace` para reemplazarla. Pero hay otra herramienta, llamada **expresión regular**, que también puede realizar estas operaciones – y muchas más.

Para demostrarlo, empezaré con un ejemplo sencillo y avanzaremos poco a poco. Supongamos, otra vez, que queremos encontrar todas las líneas que contienen una palabra concreta. Para variar, busquemos referencias al personaje titular del libro, el conde Dracula. Aquí tienes una línea que lo menciona.

```
text = "I am Dracula; and I bid you welcome, Mr. Harker, to my house."
```

Y aquí está el **patrón** que usaremos para buscar.

```
pattern = 'Dracula'
```

Un módulo llamado `re` proporciona funciones relacionadas con expresiones regulares. Podemos importarlo así y usar la función `search` para comprobar si el patrón aparece en el texto.

```
import re

result = re.search(pattern, text)
result
```

Si el patrón aparece en el texto, `search` devuelve un objeto `Match` que contiene los resultados de la búsqueda. Entre otra información, tiene una variable llamada `string` que contiene el texto en el que se buscó.

```
result.string
```

También proporciona un método llamado `group` que devuelve la parte del texto que coincidió con el patrón.

```
result.group()
```

Y proporciona un método llamado `span` que devuelve el índice en el texto donde empieza y termina el patrón.

```
result.span()
```

Si el patrón no aparece en el texto, el valor de retorno de `search` es `None`.

```
result = re.search('Count', text)
print(result)
```

Así que podemos comprobar si la búsqueda tuvo éxito comprobando si el resultado es `None`.

```
result == None
```

Juntando todo eso, aquí tienes una función que recorre las líneas del libro hasta encontrar una que coincida con el patrón dado, y devuelve el objeto `Match`.

```
def find_first(pattern):
    for line in open('pg345_cleaned.txt'):
        result = re.search(pattern, line)
        if result != None:
            return result
```

Podemos usarla para encontrar la primera mención de un personaje.

```
result = find_first('Harker')
result.string
```

Para este ejemplo, no tuvimos que usar expresiones regulares – podríamos haber hecho lo mismo más fácilmente con el operador `in`. Pero las expresiones regulares pueden hacer cosas que el operador `in` no puede.

Por ejemplo, si el patrón incluye el carácter de barra vertical, `'|'`, puede coincidir con la secuencia de la izquierda o con la secuencia de la derecha. Supongamos que queremos encontrar la primera mención de Mina Murray en el libro, pero no estamos seguros de si se la menciona por su nombre o por su apellido. Podemos usar el siguiente patrón, que coincide con cualquiera de los dos nombres.

```
pattern = 'Mina|Murray'
result = find_first(pattern)
result.string
```

Podemos usar un patrón como este para ver cuántas veces se menciona a un personaje por cualquiera de los dos nombres. Aquí tienes una función que recorre el libro y cuenta el número de líneas que coinciden con el patrón dado.

```
def count_matches(pattern):
    count = 0
    for line in open('pg345_cleaned.txt'):
        result = re.search(pattern, line)
        if result != None:
            count += 1
    return count
```

Ahora veamos cuántas veces se menciona a Mina.

```
count_matches('Mina|Murray')
```

El carácter especial `'^'` coincide con el principio de una cadena, así que podemos encontrar una línea que empieza con un patrón dado.

```
result = find_first('^Dracula')
result.string
```

Y el carácter especial `'$'` coincide con el final de una cadena, así que podemos encontrar una línea que termina con un patrón dado (ignorando el salto de línea del final).

```
result = find_first('Harker$')
result.string
```

8.9. Sustitución de cadenas

Bram Stoker nació en Irlanda, y cuando *Dracula* se publicó en 1897, vivía en Inglaterra. Así que esperaríamos que usara la ortografía británica de palabras como "centre" y "colour". Para comprobarlo, podemos usar el siguiente patrón, que coincide con "centre" o con la ortografía estadounidense "center".

```
pattern = 'cent(er|re)'
```

En este patrón, los paréntesis encierran la parte del patrón a la que se aplica la barra vertical. Así que este patrón coincide con una secuencia que empieza con `'cent'` y termina con `'er'` o con `'re'`.

```
result = find_first(pattern)
result.string
```

Como esperábamos, usó la ortografía británica.

También podemos comprobar si usó la ortografía británica de "colour". El siguiente patrón usa el carácter especial `'?'`, que significa que el carácter anterior es opcional.

```
pattern = 'colou?r'
```

Este patrón coincide con "colour" con la `'u'` o con "color" sin ella.

```
result = find_first(pattern)
line = result.string
line
```

De nuevo, como esperábamos, usó la ortografía británica.

Ahora supongamos que queremos producir una edición del libro con ortografía estadounidense. Podemos usar la función `sub` del módulo `re`, que hace **sustitución de cadenas**.

```
re.sub(pattern, 'color', line)
```

El primer argumento es el patrón que queremos encontrar y reemplazar, el segundo es aquello con lo que queremos reemplazarlo, y el tercero es la cadena en el que queremos buscar. En el resultado, puedes ver que “colour” ha sido reemplazado por “color”.

```
# I used this function to search for lines to use as examples

def all_matches(pattern):
    for line in open('pg345_cleaned.txt'):
        result = re.search(pattern, line)
        if result:
            print(line.strip())
```

```
# Here's the pattern I used (which uses some features we haven't seen)

names = r'(?<!\.\s)[A-Z][a-zA-Z]+'

all_matches(names)
```

8.10. Depuración

Cuando lees y escribes archivos, depurar puede ser complicado. Si trabajas en un Jupyter notebook, puedes usar **comandos de shell** para ayudar. Por ejemplo, para mostrar las primeras líneas de un archivo, puedes usar el comando `!head`, así:

```
!head pg345_cleaned.txt
```

El signo de exclamación inicial, `!`, indica que esto es un comando de shell, que no forma parte de Python. Para mostrar las últimas líneas, puedes usar `!tail`.

```
!tail pg345_cleaned.txt
```

Cuando trabajas con archivos grandes, depurar puede ser difícil porque puede haber demasiada salida para revisarla a mano. Una buena estrategia de depuración es empezar con solo una parte del archivo, hacer que el programa funcione, y luego ejecutarlo con el archivo completo.

Para crear un archivo pequeño que contenga parte de un archivo más grande, podemos usar `!head` de nuevo con el operador de redirección, `>`, que indica que los resultados deben

escribirse en un archivo en lugar de mostrarse.

```
!head pg345_cleaned.txt > pg345_cleaned_10_lines.txt
```

Por defecto, `!head` lee las primeras 10 líneas, pero recibe un argumento opcional que indica el número de líneas a leer.

```
!head -100 pg345_cleaned.txt > pg345_cleaned_100_lines.txt
```

Este comando de shell lee las primeras 100 líneas de `pg345_cleaned.txt` y las escribe en un archivo llamado `pg345_cleaned_100_lines.txt`.

Nota: Los comandos de shell `!head` y `!tail` no están disponibles en todos los sistemas operativos. Si no te funcionan, podemos escribir funciones similares en Python. Consulta el primer ejercicio al final de este capítulo para ver sugerencias.

8.11. Glosario

secuencia: Una colección ordenada de valores donde cada valor se identifica mediante un índice entero.

carácter: Un elemento de una cadena, incluidas letras, números y símbolos.

índice: Un valor entero usado para seleccionar un elemento en una secuencia, como un carácter en una cadena. En Python los índices empiezan desde `0`.

segmento: Una parte de una cadena especificada por un rango de índices.

cadena vacía: Una cadena que no contiene caracteres y tiene longitud `0`.

objeto: Algo a lo que una variable puede referirse. Un objeto tiene un tipo y un valor.

inmutable: Si los elementos de un objeto no se pueden cambiar, el objeto es inmutable.

invocación: Una expresión – o parte de una expresión – que llama a un método.

expresión regular: Una secuencia de caracteres que define un patrón de búsqueda.

patrón: Una regla que especifica los requisitos que una cadena debe cumplir para constituir una coincidencia.

sustitución de cadenas: Reemplazo de una cadena, o parte de una cadena, por otra cadena.

comando de shell: Una sentencia en un lenguaje de shell, que es un lenguaje usado para interactuar con un sistema operativo.

8.12. Ejercicios

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

```
download('https://raw.githubusercontent.com/AllenDowney/ThinkPython/v3/words.txt');
```

8.12.1. Pregunta a un asistente virtual

En este capítulo, apenas hemos arañado la superficie de lo que pueden hacer las expresiones regulares. Para hacerte una idea de lo que es posible, pregunta a un asistente virtual: "¿Cuáles son los caracteres especiales más comunes que se usan en las expresiones regulares de Python?"

También puedes pedir un patrón que coincida con tipos concretos de cadenas. Por ejemplo, prueba a preguntar:

- Escribe una expresión regular de Python que coincida con un número de teléfono de 10 dígitos con guiones.
- Escribe una expresión regular de Python que coincida con una dirección con un número y un nombre de calle, seguida de `ST` o `AVE`.
- Escribe una expresión regular de Python que coincida con un nombre completo con cualquier título común como `Mr` o `Mrs`, seguido de cualquier número de nombres que empiecen con mayúsculas, posiblemente con guiones entre algunos nombres.

Y si quieres ver algo más complicado, prueba a pedir una expresión regular que coincida con cualquier URL legal.

Una expresión regular a menudo tiene la letra `r` antes de la comilla, lo que indica que es una "cadena sin procesar". Para más información, pregunta a un asistente virtual: "¿Qué es una cadena sin procesar en Python?"

```
from doctest import run_docstring_examples

def run_doctests(func):
    run_docstring_examples(func, globals(), name=func.__name__)
```

8.12.2. Ejercicio

Mira si puedes escribir una función que haga lo mismo que el comando de shell `!head`. Debe recibir como argumentos el nombre de un archivo que leer, el número de líneas que leer y el nombre del archivo donde escribir las líneas. Si el tercer parámetro es `None`, debe mostrar las líneas en lugar de escribirlas en un archivo.

Considera pedir ayuda a un asistente virtual, pero si lo haces, dile que no use una sentencia `with` ni una sentencia `try`.

Puedes usar los siguientes ejemplos para probar tu función.

```
head('pg345_cleaned.txt', 10)
```

```
head('pg345_cleaned.txt', 100, 'pg345_cleaned_100_lines.txt')
```

```
!tail pg345_cleaned_100_lines.txt
```

8.12.3. Ejercicio

"Wordle" es un juego de palabras online donde el objetivo es adivinar una palabra de cinco letras en seis intentos o menos. Cada intento tiene que ser reconocido como una palabra, sin incluir nombres propios. Después de cada intento, obtienes información sobre cuáles de las letras que adivinaste aparecen en la palabra objetivo, y cuáles están en la posición correcta.

Por ejemplo, supongamos que la palabra objetivo es `MOWER` y que intentas `TRIED`. Aprenderías que `E` está en la palabra y en la posición correcta, `R` está en la palabra pero no en la posición correcta, y `T`, `I` y `D` no están en la palabra.

Como ejemplo distinto, supongamos que has intentado las palabras `SPADE` y `CLERK`, y has aprendido que `E` está en la palabra, pero no en ninguna de esas posiciones, y que ninguna de las otras letras aparece en la palabra. De las palabras de la lista, ¿cuántas podrían ser la palabra objetivo? Escribe una función llamada `check_word` que reciba una palabra de cinco letras y compruebe si podría ser la palabra objetivo, dadas estas conjeturas.

Puedes usar cualquiera de las funciones del capítulo anterior, como `uses_any`.

```
def uses_any(word, letters):
    for letter in word.lower():
        if letter in letters.lower():
            return True
    return False
```

Puedes usar el siguiente bucle para probar tu función.

```
for line in open('words.txt'):
    word = line.strip()
    if len(word) == 5 and check_word(word):
        print(word)
```

8.12.4. Ejercicio

Continuando con el ejercicio anterior, supongamos que intentas la palabra `TOTEM` y aprendes que la `E` *todavía* no está en el lugar correcto, pero la `M` sí. ¿Cuántas palabras quedan?

8.12.5. Ejercicio

The Count of Monte Cristo es una novela de Alexandre Dumas que se considera un clásico. Sin embargo, en la introducción de una traducción inglesa del libro, el escritor Umberto Eco confiesa que le pareció "una de las novelas peor escritas de todos los tiempos".

En particular, dice que es "desvergonzada en su repetición del mismo adjetivo", y menciona en particular el número de veces que "sus personajes se estremecen o palidecen".

Para ver si su objeción es válida, contemos el número de líneas que contienen la palabra `pale` en cualquier forma, incluidas `pale`, `pales`, `paled` y `paleness`, así como la palabra relacionada `pallor`. Usa una sola expresión regular que coincida con cualquiera de estas palabras. Como desafío adicional, asegúrate de que no coincida con otras palabras, como `impale` – quizá quieras pedir ayuda a un asistente virtual.

La siguiente celda descarga el libro de Project Gutenberg https://www.gutenberg.org/ebooks/1184.

```
import os

if not os.path.exists('pg1184.txt'):
    !wget https://www.gutenberg.org/cache/epub/1184/pg1184.txt
```

La siguiente celda ejecuta una función que lee el archivo de Project Gutenberg y escribe un archivo que contiene solo el texto del libro, no la información añadida sobre el libro.

```
def clean_file(input_file, output_file):
    reader = open(input_file)
    writer = open(output_file, 'w')

    for line in reader:
        if is_special_line(line):
            break

    for line in reader:
        if is_special_line(line):
            break
        writer.write(line)

    reader.close()
    writer.close()

clean_file('pg1184.txt', 'pg1184_cleaned.txt')
```

Según este recuento, estas palabras aparecen en `223` líneas del libro, así que el señor Eco quizá tenga razón.

Think Python: 3rd Edition

Traducción al español por midudev (Miguel Ángel Durán).

Puedes pedir las versiones impresa y ebook de *Think Python 3e* en Bookshop.org y Amazon.

9. Listas

Este capítulo presenta uno de los tipos integrados más útiles de Python: las listas. También aprenderás más sobre objetos y lo que puede ocurrir cuando varias variables se refieren al mismo objeto.

En los ejercicios al final del capítulo, construiremos una lista de palabras y la usaremos para buscar palabras especiales como palíndromos y anagramas.

9.1. Una lista es una secuencia

Como un string, una **lista** es una secuencia de valores. En un string, los valores son caracteres; en una lista, pueden ser de cualquier tipo. Los valores en una lista se llaman **elementos**.

Hay varias formas de crear una nueva lista; la más simple es encerrar los elementos entre corchetes (`[` y `]`). Por ejemplo, aquí tienes una lista de dos enteros.

```
numbers = [42, 123]
```

Y aquí tienes una lista de tres strings.

```
cheeses = ['Cheddar', 'Edam', 'Gouda']
```

Los elementos de una lista no tienen que ser del mismo tipo. La siguiente lista contiene un string, un float, un entero e incluso otra lista.

```
t = ['spam', 2.0, 5, [10, 20]]
```

Una lista dentro de otra lista está **anidada**.

Una lista que no contiene elementos se llama lista vacía; puedes crear una con corchetes vacíos, `[]`.

```
empty = []
```

La función `len` devuelve la longitud de una lista.

```
len(cheeses)
```

La longitud de una lista vacía es `0`.

```
len(empty)
```

La siguiente figura muestra el diagrama de estado de `cheeses`, `numbers` y `empty`.

```
from diagram import make_list, Binding, Value

list1 = make_list(cheeses, dy=-0.3, offsetx=0.17)
binding1 = Binding(Value('cheeses'), list1)

list2 = make_list(numbers, dy=-0.3, offsetx=0.17)
binding2 = Binding(Value('numbers'), list2)

list3 = make_list(empty, dy=-0.3, offsetx=0.1)
binding3 = Binding(Value('empty'), list3)
```

```
from diagram import diagram, adjust, Bbox

width, height, x, y = [3.66, 1.58, 0.45, 1.2]
ax = diagram(width, height)
bbox1 = binding1.draw(ax, x, y)
bbox2 = binding2.draw(ax, x+2.25, y)
bbox3 = binding3.draw(ax, x+2.25, y-1.0)

bbox = Bbox.union([bbox1, bbox2, bbox3])
#adjust(x, y, bbox)
```

Las listas se representan con cajas con la palabra “lista” fuera y los elementos numerados de la lista dentro.

9.2. Las listas son mutables

Para leer un elemento de una lista, podemos usar el operador de corchetes. El índice del primer elemento es `0`.

```
cheeses[0]
```

A diferencia de los strings, las listas son mutables. Cuando el operador de corchetes aparece en el lado izquierdo de una asignación, identifica el elemento de la lista al que se asignará el valor.

```
numbers[1] = 17
numbers
```

El segundo elemento de `numbers`, que antes era `123`, ahora es `17`.

Los índices de lista funcionan igual que los índices de string:

- Cualquier expresión entera puede usarse como índice.
- Si intentas leer o escribir un elemento que no existe, obtienes un `IndexError`.
- Si un índice tiene un valor negativo, cuenta hacia atrás desde el final de la lista.

El operador `in` funciona con listas: comprueba si un elemento dado aparece en algún lugar de la lista.

```
'Edam' in cheeses
```

```
'Wensleydale' in cheeses
```

Aunque una lista puede contener otra lista, la lista anidada sigue contando como un solo elemento; por eso, en la siguiente lista, solo hay cuatro elementos.

```
t = ['spam', 2.0, 5, [10, 20]]
len(t)
```

Y `10` no se considera un elemento de `t` porque es un elemento de una lista anidada, no de `t`.

```
10 in t
```

9.3. Porciones de lista

El operador de porción funciona con listas igual que con strings. El siguiente ejemplo selecciona el segundo y el tercer elemento de una lista de cuatro letras.

```
letters = ['a', 'b', 'c', 'd']
letters[1:3]
```

Si omites el primer índice, la porción empieza al principio.

```
letters[:2]
```

Si omites el segundo, la porción llega hasta el final.

```
letters[2:]
```

Así que, si omites ambos, la porción es una copia de toda la lista.

```
letters[:]
```

Otra forma de copiar una lista es usar la función `list`.

```
list(letters)
```

Como `list` es el nombre de una función integrada, deberías evitar usarlo como nombre de variable.

9.4. Operaciones con listas

El operador `+` concatena listas.

```
t1 = [1, 2]
t2 = [3, 4]
t1 + t2
```

El operador `*` repite una lista un número determinado de veces.

```
['spam'] * 4
```

Ningún otro operador matemático funciona con listas, pero la función integrada `sum` suma los elementos.

```
sum(t1)
```

Y `min` y `max` encuentran el elemento más pequeño y el más grande.

```
min(t1)
```

```
max(t2)
```

9.5. Métodos de lista

Python proporciona métodos que operan sobre listas. Por ejemplo, `append` añade un nuevo elemento al final de una lista:

```
letters.append('e')
letters
```

`extend` toma una lista como argumento y añade todos sus elementos:

```
letters.extend(['f', 'g'])
letters
```

Hay dos métodos que eliminan elementos de una lista. Si conoces el índice del elemento que quieres, puedes usar `pop`.

```
t = ['a', 'b', 'c']
t.pop(1)
```

El valor de retorno es el elemento que se eliminó. Y podemos confirmar que la lista se ha modificado.

```
t
```

Si conoces el elemento que quieres eliminar (pero no el índice), puedes usar `remove`:

```
t = ['a', 'b', 'c']
t.remove('b')
```

El valor de retorno de `remove` es `None`. Pero podemos confirmar que la lista se ha modificado.

```
t
```

Si el elemento que pides no está en la lista, eso es un ValueError.

```
t.remove('d')
```

9.6. Listas y strings

Un string es una secuencia de caracteres y una lista es una secuencia de valores, pero una lista de caracteres no es lo mismo que un string. Para convertir un string en una lista de caracteres, puedes usar la función `list`.

```
s = 'spam'
t = list(s)
t
```

La función `list` divide un string en letras individuales. Si quieres dividir un string en palabras, puedes usar el método `split`:

```
s = 'pining for the fjords'
t = s.split()
t
```

Un argumento opcional llamado **delimitador** especifica qué caracteres se usan como límites entre palabras. El siguiente ejemplo usa un guion como delimitador.

```
s = 'ex-parrot'
t = s.split('-')
t
```

Si tienes una lista de strings, puedes concatenarlos en un solo string usando `join`. `join` es un método de string, así que tienes que invocarlo sobre el delimitador y pasar la lista como argumento.

```
delimiter = ' '
t = ['pining', 'for', 'the', 'fjords']
s = delimiter.join(t)
s
```

En este caso el delimitador es un carácter de espacio, así que `join` pone un espacio entre las palabras. Para unir strings sin espacios, puedes usar el string vacío, `''`, como delimitador.

9.7. Recorrer una lista con un bucle

Puedes usar una sentencia `for` para recorrer los elementos de una lista.

```
for cheese in cheeses:
    print(cheese)
```

Por ejemplo, después de usar `split` para crear una lista de palabras, podemos usar `for` para recorrerlas.

```
s = 'pining for the fjords'

for word in s.split():
    print(word)
```

Un bucle `for` sobre una lista vacía nunca ejecuta las sentencias indentadas.

```
for x in []:
    print('This never happens.')
```

9.8. Ordenar listas

Python proporciona una función integrada llamada `sorted` que ordena los elementos de una lista.

```
scramble = ['c', 'a', 'b']
sorted(scramble)
```

La lista original no cambia.

```
scramble
```

`sorted` funciona con cualquier tipo de secuencia, no solo con listas. Así que podemos ordenar las letras de un string así.

```
sorted('letters')
```

El resultado es una lista. Para convertir la lista en un string, podemos usar `join`.

```
''.join(sorted('letters'))
```

Con un string vacío como delimitador, los elementos de la lista se unen sin nada entre ellos.

9.9. Objetos y valores

Si ejecutamos estas sentencias de asignación:

```
a = 'banana'
b = 'banana'
```

Sabemos que `a` y `b` se refieren ambos a un string, pero no sabemos si se refieren al *mismo* string. Hay dos estados posibles, mostrados en la siguiente figura.

```
from diagram import Frame, Stack

s = 'banana'
bindings = [Binding(Value(name), Value(repr(s))) for name in 'ab']
frame1 = Frame(bindings, dy=-0.25)

binding1 = Binding(Value('a'), Value(repr(s)), dy=-0.11)
binding2 = Binding(Value('b'), draw_value=False, dy=0.11)
frame2 = Frame([binding1, binding2], dy=-0.25)

stack = Stack([frame1, frame2], dx=1.7, dy=0)
```

```
width, height, x, y = [2.85, 0.76, 0.17, 0.51]
ax = diagram(width, height)
bbox = stack.draw(ax, x, y)
# adjust(x, y, bbox)
```

En el diagrama de la izquierda, `a` y `b` se refieren a dos objetos diferentes que tienen el mismo valor. En el diagrama de la derecha, se refieren al mismo objeto. Para comprobar si dos variables se refieren al mismo objeto, puedes usar el operador `is`.

```
a = 'banana'
b = 'banana'
a is b
```

En este ejemplo, Python solo creó un objeto string, y tanto `a` como `b` se refieren a él. Pero cuando creas dos listas, obtienes dos objetos.

```
a = [1, 2, 3]
b = [1, 2, 3]
a is b
```

Así que el diagrama de estado se ve así.

```
t = [1, 2, 3]
binding1 = Binding(Value('a'), Value(repr(t)))
binding2 = Binding(Value('b'), Value(repr(t)))
frame = Frame([binding1, binding2], dy=-0.25)
```

```
width, height, x, y = [1.16, 0.76, 0.21, 0.51]
ax = diagram(width, height)
bbox = frame.draw(ax, x, y)
# adjust(x, y, bbox)
```

En este caso diríamos que las dos listas son **equivalentes**, porque tienen los mismos elementos, pero no **idénticos**, porque no son el mismo objeto. Si dos objetos son idénticos, también son equivalentes, pero si son equivalentes, no necesariamente son idénticos.

9.10. Aliasing

Si `a` se refiere a un objeto y asignas `b = a`, entonces ambas variables se refieren al mismo objeto.

```
a = [1, 2, 3]
b = a
b is a
```

Así que el diagrama de estado se ve así.

```
t = [1, 2, 3]
binding1 = Binding(Value('a'), Value(repr(t)), dy=-0.11)
binding2 = Binding(Value('b'), draw_value=False, dy=0.11)
frame = Frame([binding1, binding2], dy=-0.25)
```

```
width, height, x, y = [1.11, 0.81, 0.17, 0.56]
ax = diagram(width, height)
bbox = frame.draw(ax, x, y)
# adjust(x, y, bbox)
```

La asociación de una variable con un objeto se llama **referencia**. En este ejemplo, hay dos referencias al mismo objeto.

Un objeto con más de una referencia tiene más de un nombre, así que decimos que el objeto está **aliased**. Si el aliased objeto es mutable, los cambios hechos con un nombre afectan al otro. En este ejemplo, si cambiamos el objeto al que se refiere `b`, también estamos cambiando el objeto al que se refiere `a`.

```
b[0] = 5
a
```

Así que diríamos que `a` "ve" este cambio. Aunque este comportamiento puede ser útil, es propenso a errores. En general, es más seguro evitar el aliasing cuando trabajas con objetos mutables.

Para objetos inmutables como strings, el aliasing no supone tanto problema. En este ejemplo:

```
a = 'banana'
b = 'banana'
```

Casi nunca importa si `a` y `b` se refieren al mismo string o no.

9.11. Listas como argumentos

Cuando pasas una lista a una función, la función recibe una referencia a la lista. Si la función modifica la lista, el código que llama ve el cambio. Por ejemplo, `pop_first` usa el método de lista `pop` para eliminar el primer elemento de una lista.

```
def pop_first(lst):
    return lst.pop(0)
```

Podemos usarla así.

```
letters = ['a', 'b', 'c']
pop_first(letters)
```

El valor de retorno es el primer elemento, que se ha eliminado de la lista, como podemos ver al mostrar la lista modificada.

```
letters
```

En este ejemplo, el parameter `lst` y la variable `letters` son aliases del mismo objeto, así que el diagrama de estado se ve así:

```
lst = make_list('abc', dy=-0.3, offsetx=0.1)
binding1 = Binding(Value('letters'), draw_value=False)
frame1 = Frame([binding1], name='__main__', loc='left')

binding2 = Binding(Value('lst'), draw_value=False, dx=0.61, dy=0.35)
frame2 = Frame([binding2], name='pop_first', loc='left', offsetx=0.08)

stack = Stack([frame1, frame2], dx=-0.3, dy=-0.5)
```

```
width, height, x, y = [2.04, 1.24, 1.06, 0.85]
ax = diagram(width, height)
bbox1 = stack.draw(ax, x, y)
bbox2 = lst.draw(ax, x+0.5, y)
bbox = Bbox.union([bbox1, bbox2])
adjust(x, y, bbox)
```

Pasar una referencia a un objeto como argumento a una función crea una forma de aliasing. Si la función modifica el objeto, esos cambios persisten después de que la función termine.

9.12. Crear una lista de palabras

En el capítulo anterior, leímos el archivo `words.txt` y buscamos palabras con ciertas propiedades, como usar la letra `e`. Pero leímos el archivo completo muchas veces, lo cual no es eficiente. Es mejor leer el archivo una vez y poner las palabras en una lista. El siguiente bucle muestra cómo hacerlo.

```
download('https://raw.githubusercontent.com/AllenDowney/ThinkPython/v3/words.txt');
```

```
word_list = []

for line in open('words.txt'):
    word = line.strip()
    word_list.append(word)

len(word_list)
```

Antes del bucle, `word_list` se inicializa con una lista vacía. Cada vez que pasa por el bucle, el método `append` añade una palabra al final. Cuando termina el bucle, hay más de 113 000 palabras en la lista.

Otra forma de hacer lo mismo es usar `read` para leer todo el archivo en un string.

```
string = open('words.txt').read()
len(string)
```

El resultado es un único string con más de un millón de caracteres. Podemos usar el método `split` para dividirlo en una lista de palabras.

```
word_list = string.split()
len(word_list)
```

Ahora, para comprobar si un string aparece en la lista, podemos usar el operador `in`. Por ejemplo, `'demotic'` está en la lista.

```
'demotic' in word_list
```

Pero `'contrafibularities'` no está.

```
'contrafibularities' in word_list
```

Y tengo que decir que eso me deja anaspeptic.

9.13. Depuración

Ten en cuenta que la mayoría de métodos de lista modifican el argumento y devuelven `None`. Esto es lo contrario de los métodos de string, que devuelven un nuevo string y dejan el original intacto.

Si estás acostumbrado a escribir código con strings así:

```
word = 'plumage!'
word = word.strip('!')
word
```

Es tentador escribir código con listas así:

```
t = [1, 2, 3]
t = t.remove(3)            # WRONG!
```

`remove` modifica la lista y devuelve `None`, así que la siguiente operación que realices con `t` probablemente fallará.

```
t.remove(2)
```

Este mensaje de error requiere algo de explicación. Un **atributo** de un objeto es una variable o método asociado con él. En este caso, el valor de `t` es `None`, que es un objeto `NoneType`, que no tiene un atributo llamado `remove`, así que el resultado es un `AttributeError`.

Si ves un mensaje de error como este, deberías mirar hacia atrás en el programa y ver si podrías haber llamado incorrectamente a un método de lista.

9.14. Glosario

lista: Un objeto que contiene una secuencia de valores.

elemento: Uno de los valores de una lista u otra secuencia.

lista anidada: Una lista que es un elemento de otra lista.

delimitador: Un carácter o string usado para indicar dónde debe dividirse un string.

equivalentes: Tener el mismo valor.

idénticos: Ser el mismo objeto (lo que implica equivalencia).

referencia: La asociación entre una variable y su valor.

aliased: Si hay más de una variable que se refiere a un objeto, el objeto está aliased.

atributo: Uno de los valores con nombre asociados con un objeto.

9.15. Ejercicios

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

9.15.1. Pregunta a un asistente virtual

En este capítulo, usé las palabras “contrafibularities” y “anaspeptic”, pero en realidad no son palabras en inglés. Se usaron en la serie de televisión británica *Black Adder*, temporada 3, episodio 2, “Ink and Incapability”.

Sin embargo, cuando pregunté a ChatGPT 3.5 (versión del 3 de agosto de 2023) de dónde venían esas palabras, al principio afirmó que eran de Monty Python, y después afirmó que eran de la obra de Tom Stoppard *Rosencrantz and Guildenstern Are Dead*.

Si preguntas ahora, podrías obtener resultados diferentes. Pero este ejemplo es un recordatorio de que los asistentes virtuales no siempre son precisos, así que deberías comprobar si los resultados son correctos. A medida que ganes experiencia, desarrollarás intuición sobre qué preguntas pueden responder de forma fiable los asistentes virtuales. En este ejemplo, una búsqueda web convencional puede identificar rápidamente el origen de estas palabras.

Si te quedas atascado en alguno de los ejercicios de este capítulo, considera pedir ayuda a un asistente virtual. Si obtienes un resultado que usa características que todavía no hemos aprendido, puedes asignar un “role” al asistente virtual.

Por ejemplo, antes de hacer una pregunta prueba a escribir “Role: Basic Python Programming Instructor”. Después de eso, las respuestas que obtengas deberían usar solo características básicas. Si todavía ves características que no hemos aprendido, puedes continuar con “Can you write that using only basic Python features?”

9.15.2. Ejercicio

Dos palabras son anagramas si puedes reordenar las letras de una para escribir la otra. Por ejemplo, `tops` es un anagrama de `stop`.

Una forma de comprobar si dos palabras son anagramas es ordenar las letras de ambas palabras. Si las listas de letras ordenadas son iguales, las palabras son anagramas.

Escribe una función llamada `is_anagram` que tome dos strings y devuelva `True` si son anagramas.

Para empezar, aquí tienes un esquema de la función con doctests.

```
def is_anagram(word1, word2):
    """Checks whether two words are anagrams.

    >>> is_anagram('tops', 'stop')
    True
    >>> is_anagram('skate', 'takes')
    True
    >>> is_anagram('tops', 'takes')
    False
    >>> is_anagram('skate', 'stop')
    False
    """
    return None
```

Puedes usar `doctest` para probar tu función.

```
from doctest import run_docstring_examples

def run_doctests(func):
    run_docstring_examples(func, globals(), name=func.__name__)

run_doctests(is_anagram)
```

Usando tu función y la lista de palabras, encuentra todos los anagramas de `takes`.

9.15.3. Ejercicio

Python proporciona una función integrada llamada `reversed` que toma como argumento una secuencia de elementos, como una lista o un string, y devuelve un objeto `reversed` que contiene los elementos en orden inverso.

```
reversed('parrot')
```

Si quieres los elementos invertidos en una lista, puedes usar la función `list`.

```
list(reversed('parrot'))
```

O si los quieres en un string, puedes usar el método `join`.

```
''.join(reversed('parrot'))
```

Así podemos escribir una función que invierte una palabra así.

```
def reverse_word(word):
    return ''.join(reversed(word))
```

Un palíndromo es una palabra que se escribe igual hacia atrás y hacia adelante, como “noon” y “rotator”. Escribe una función llamada `is_palindrome` que tome un string argumento y devuelva `True` si es un palíndromo y `False` en caso contrario.

Aquí tienes un esquema de la función con doctests que puedes usar para comprobar tu función.

```
def is_palindrome(word):
    """Check if a word is a palindrome.

    >>> is_palindrome('bob')
    True
    >>> is_palindrome('alice')
    False
    >>> is_palindrome('a')
    True
    >>> is_palindrome('')
    True
    """
    return False
```

```
run_doctests(is_palindrome)
```

Puedes usar el siguiente bucle para encontrar todos los palíndromos en la lista de palabras con al menos 7 letras.

```
for word in word_list:
    if len(word) >= 7 and is_palindrome(word):
        print(word)
```

9.15.4. Ejercicio

Escribe una función llamada `reverse_sentence` que tome como argumento un string que contiene cualquier número de palabras separadas por espacios. Debe devolver un nuevo string que contiene las mismas palabras en orden inverso. Por ejemplo, si el argumento es "Reverse this sentence", el resultado debería ser "Sentence this reverse".

Pista: Puedes usar los métodos `capitalize` para poner en mayúscula la primera palabra y convertir las demás palabras a minúsculas.

Para empezar, aquí tienes un esquema de la función con doctests.

```
def reverse_sentence(input_string):
    '''Reverse the words in a string and capitalize the first.

    >>> reverse_sentence('Reverse this sentence')
    'Sentence this reverse'

    >>> reverse_sentence('Python')
    'Python'

    >>> reverse_sentence('')
    ''

    >>> reverse_sentence('One for all and all for one')
    'One for all and all for one'
    '''
    return None
```

```
run_doctests(reverse_sentence)
```

9.15.5. Ejercicio

Escribe una función llamada `total_length` que tome una lista de strings y devuelva la longitud total de los strings. La longitud total de las palabras en `word_list` debería ser $902{,}728$.

Think Python: 3rd Edition

Traducción al español por midudev (Miguel Ángel Durán).

Puedes pedir las versiones impresa y ebook de *Think Python 3e* en Bookshop.org y Amazon.

10. Diccionarios

Este capítulo presenta un tipo integrado llamado diccionario. Es una de las mejores características de Python, y el bloque de construcción de muchos algoritmos eficientes y elegantes.

Usaremos diccionarios para calcular el número de palabras únicas en un libro y el número de veces que aparece cada una. Y en los ejercicios, usaremos diccionarios para resolver juegos de palabras.

10.1. Un diccionario es un mapeo

Un **diccionario** es como una lista, pero más general. En una lista, los índices tienen que ser enteros; en un diccionario pueden ser de (casi) cualquier tipo. Por ejemplo, supongamos que hacemos una lista de palabras numéricas, así.

```
lst = ['zero', 'one', 'two']
```

Podemos usar un entero como índice para obtener la palabra correspondiente.

```
lst[1]
```

Pero supongamos que queremos ir en la otra dirección, y buscar una palabra para obtener el entero correspondiente. No podemos hacerlo con una lista, pero sí con un diccionario. Empezaremos creando un diccionario vacío y asignándolo a `numbers`.

```
numbers = {}
numbers
```

Las llaves, `{}`, representan un diccionario vacío. Para añadir elementos al diccionario, usaremos corchetes.

```
numbers['zero'] = 0
```

Esta asignación añade al diccionario un **elemento**, que representa la asociación de una **clave** y un **valor**. En este ejemplo, la clave es el string `'zero'` y el valor es el entero `0`. Si mostramos el diccionario, vemos que contiene un elemento, que contiene una clave y un valor separados por dos puntos, `:`.

```
numbers
```

Podemos añadir más elementos así.

```
numbers['one'] = 1
numbers['two'] = 2
numbers
```

Ahora el diccionario contiene tres elementos.

Para buscar una clave y obtener el valor correspondiente, usamos el operador de corchetes.

```
numbers['two']
```

Si la clave no está en el diccionario, obtenemos un `KeyError`.

```
numbers['three']
```

La función `len` funciona con diccionarios; devuelve el número de elementos.

```
len(numbers)
```

En lenguaje matemático, un diccionario representa un **mapeo** de claves a valores, así que también puedes decir que cada clave "mapea a" un valor. En este ejemplo, cada palabra numérica mapea al entero correspondiente.

La siguiente figura muestra el diagrama de estado de `numbers`.

```
from diagram import make_dict, Binding, Value

d1 = make_dict(numbers, dy=-0.3, offsetx=0.37)
binding1 = Binding(Value('numbers'), d1)
```

```
from diagram import diagram, adjust, Bbox

width, height, x, y = [1.83, 1.24, 0.49, 0.85]
ax = diagram(width, height)
bbox = binding1.draw(ax, x, y)
# adjust(x, y, bbox)
```

Un diccionario se representa con una caja con la palabra "dict" fuera y los elementos dentro. Cada elemento se representa con una clave y una flecha que apunta a un valor. Las comillas indican que las claves aquí son strings, no nombres de variables.

10.2. Crear diccionarios

En la sección anterior creamos un diccionario vacío y añadimos elementos uno por uno usando el operador de corchetes. En su lugar, podríamos haber creado el diccionario de una vez así.

```
numbers = {'zero': 0, 'one': 1, 'two': 2}
```

Cada elemento consiste en una clave y un valor separados por dos puntos. Los elementos se separan con comas y se encierran entre llaves.

Otra forma de crear un diccionario es usar la función `dict`. Podemos crear un diccionario vacío así.

```
empty = dict()
empty
```

Y podemos hacer una copia de un diccionario así.

```
numbers_copy = dict(numbers)
numbers_copy
```

A menudo es útil hacer una copia antes de realizar operaciones que modifican diccionarios.

10.3. El operador in

El operador `in` también funciona con diccionarios; te dice si algo aparece como *clave* en el diccionario.

```
'one' in numbers
```

El operador `in` *no* comprueba si algo aparece como valor.

```
1 in numbers
```

Para ver si algo aparece como valor en un diccionario, puedes usar el método `values`, que devuelve una secuencia de valores, y luego usar el operador `in`.

```
1 in numbers.values()
```

Los elementos de un diccionario de Python se almacenan en una **hash table**, que es una forma de organizar datos con una propiedad notable: el operador `in` tarda aproximadamente la misma cantidad de tiempo sin importar cuántos elementos haya en el diccionario. Eso hace posible escribir algunos algoritmos notablemente eficientes.

```
download('https://raw.githubusercontent.com/AllenDowney/ThinkPython/v3/words.txt');
```

Para demostrarlo, compararemos dos algoritmos para encontrar pares de palabras donde una es la inversa de la otra, como `stressed` y `desserts`. Empezaremos leyendo la lista de palabras.

```
word_list = open('words.txt').read().split()
len(word_list)
```

Y aquí está `reverse_word` del capítulo anterior.

```
def reverse_word(word):
    return ''.join(reversed(word))
```

La siguiente función recorre las palabras de la lista. Para cada una, invierte las letras y luego comprueba si la palabra invertida está en la palabra lista.

```
def too_slow():
    count = 0
    for word in word_list:
        if reverse_word(word) in word_list:
            count += 1
    return count
```

Esta función tarda más de un minuto en ejecutarse. El problema es que el operador `in` comprueba las palabras de la lista una por una, empezando por el principio. Si no encuentra lo que busca, que ocurre la mayor parte del tiempo, tiene que buscar hasta el final.

Para medir cuánto tarda una función, podemos usar `%time`, que es uno de los "built-in magic commands" de Jupyter. Estos comandos no forman parte del lenguaje Python, así que puede que no funcionen en otros entornos de desarrollo.

```
# %time too_slow()
```

Y el operador `in` está dentro del bucle, así que se ejecuta una vez por cada palabra. Como hay más de 100 000 palabras en la lista, y para cada una comprobamos más de 100 000 palabras, el número total de comparaciones es el número de palabras al cuadrado – aproximadamente –, que es casi 13 mil millones.

```
len(word_list)**2
```

Podemos hacer que esta función sea mucho más rápida con un diccionario. El siguiente bucle crea un diccionario que contiene las palabras como claves.

```
word_dict = {}
for word in word_list:
    word_dict[word] = 1
```

Los valores en `word_dict` son todos `1`, pero podrían ser cualquier cosa, porque nunca los vamos a buscar; solo usaremos este diccionario para comprobar si existe una clave.

Ahora aquí tienes una versión de la función anterior que reemplaza `word_list` por `word_dict`.

```
def much_faster():
    count = 0
    for word in word_dict:
        if reverse_word(word) in word_dict:
            count += 1
    return count
```

Esta función tarda menos de una centésima de segundo, así que es unas 10 000 veces más rápida que la versión anterior.

En general, el tiempo que se tarda en encontrar un elemento en una lista es proporcional a la longitud de la lista. El tiempo que se tarda en encontrar una clave en un diccionario es casi constante, independientemente del número de elementos.

```
%time much_faster()
```

10.4. Una colección de contadores

Supongamos que recibes un string y quieres contar cuántas veces aparece cada letra. Un diccionario es una buena herramienta para este trabajo. Empezaremos con un diccionario vacío.

```
counter = {}
```

Mientras recorremos las letras del string, supongamos que vemos la letra `'a'` por primera vez. Podemos añadirla al diccionario así.

```
counter['a'] = 1
```

El valor `1` indica que hemos visto la letra una vez. Más tarde, si volvemos a ver la misma letra, podemos incrementar el contador así.

```
counter['a'] += 1
```

Ahora el valor asociado con `'a'` es `2`, porque hemos visto la letra dos veces.

```
counter
```

La siguiente función usa estas características para contar el número de veces que aparece cada letra en un string.

```
def value_counts(string):
    counter = {}
    for letter in string:
        if letter not in counter:
            counter[letter] = 1
        else:
            counter[letter] += 1
    return counter
```

Cada vez que pasa por el bucle, si `letter` no está en el diccionario, creamos un nuevo elemento con clave `letter` y valor `1`. Si `letter` ya está en el diccionario, incrementamos el valor asociado con `letter`.

Aquí tienes un ejemplo.

```
counter = value_counts('brontosaurus')
counter
```

Los elementos de `counter` muestran que la letra `'b'` aparece una vez, `'r'` aparece dos veces, y así sucesivamente.

10.5. Looping y diccionarios

Si usas un diccionario en una sentencia `for`, recorre las claves del diccionario. Para demostrarlo, hagamos un diccionario que cuente las letras de `'banana'`.

```
counter = value_counts('banana')
counter
```

El siguiente bucle imprime las claves, que son las letras.

```
for key in counter:
    print(key)
```

Para imprimir los valores, podemos usar el método `values`.

```
for value in counter.values():
    print(value)
```

Para imprimir las claves y los valores, podemos recorrer las claves y buscar los valores correspondientes.

```
for key in counter:
    value = counter[key]
    print(key, value)
```

En el próximo capítulo, veremos una forma más concisa de hacer lo mismo.

10.6. Listas y diccionarios

Puedes poner una lista en un diccionario como valor. Por ejemplo, aquí tienes un diccionario que mapea el número `4` a una lista de cuatro letras.

```
d = {4: ['r', 'o', 'u', 's']}
d
```

Pero no puedes poner una lista en un diccionario como clave. Esto es lo que ocurre si lo intentamos.

```
letters = list('abcd')
d[letters] = 4
```

Mencioné antes que los diccionarios usan hash tables, y eso significa que las claves tienen que ser **hasheable**.

Un **hash** es una función que toma un valor (de cualquier tipo) y devuelve un entero. Los diccionarios usan estos enteros, llamados hash valores, para almacenar y buscar claves.

Este sistema solo funciona si una clave es inmutable, de modo que su hash valor siempre sea el mismo. Pero si una clave es mutable, su hash valor podría cambiar, y el diccionario no funcionaría. Por eso las claves tienen que ser hasheables, y por eso tipos mutables como las listas no lo son.

Como los diccionarios son mutables, tampoco pueden usarse como claves. Pero *sí* pueden usarse como valores.

10.7. Acumular una lista

Para muchas tareas de programación, es útil recorrer una lista o diccionario mientras se construye otro. Como ejemplo, recorreremos las palabras en `word_dict` y haremos una lista de palíndromos, es decir, palabras que se escriben igual hacia atrás y hacia adelante, como "noon" y "rotator".

En el capítulo anterior, uno de los ejercicios te pidió escribir una función que comprueba si una palabra es un palíndromo. Aquí tienes una solución que usa `reverse_word`.

```
def is_palindrome(word):
    """Check if a word is a palindrome."""
    return reverse_word(word) == word
```

Si recorremos las palabras en `word_dict`, podemos contar el número de palíndromos así.

```
count = 0

for word in word_dict:
    if is_palindrome(word):
        count +=1

count
```

A estas alturas, este patrón resulta familiar.

- Antes del bucle, `count` se inicializa a `0`.

- Dentro del bucle, si `word` es un palíndromo, incrementamos `count`.
- Cuando termina el bucle, `count` contiene el número total de palíndromos.

Podemos usar un patrón similar para hacer una lista de palíndromos.

```
palindromes = []

for word in word_dict:
    if is_palindrome(word):
        palindromes.append(word)

palindromes[:10]
```

Así es como funciona:

- Antes del bucle, `palindromes` se inicializa con una lista vacía.
- Dentro del bucle, si `word` es un palíndromo, lo añadimos al final de `palindromes`.
- Cuando termina el bucle, `palindromes` es una lista de palíndromos.

En este bucle, `palindromes` se usa como **accumulator**, que es una variable que recoge o acumula datos durante una computación.

Ahora supongamos que queremos seleccionar solo palíndromos con siete o más letras. Podemos recorrer `palindromes` y hacer una nueva lista que contenga solo palíndromos largos.

```
long_palindromes = []

for word in palindromes:
    if len(word) >= 7:
        long_palindromes.append(word)

long_palindromes
```

Recorrer una lista así, seleccionando algunos elementos y omitiendo otros, se llama **filtrado**.

10.8. Memos

Si ejecutaste la función `fibonacci` del Capítulo 6, quizá notaste que cuanto mayor es el argumento que proporcionas, más tarda la función en ejecutarse.

```
def fibonacci(n):
    if n == 0:
        return 0

    if n == 1:
        return 1

    return fibonacci(n-1) + fibonacci(n-2)
```

Además, el tiempo de ejecución aumenta rápidamente. Para entender por qué, considera la siguiente figura, que muestra el **call graph** de `fibonacci` con `n=4`:

```
from diagram import make_binding, Frame, Arrow

bindings = [make_binding('n', i) for i in range(5)]
frames = [Frame([binding]) for binding in bindings]
```

```
arrowprops = dict(arrowstyle="-", color='gray', alpha=0.5, ls='-', lw=0.5)

def left_arrow(ax, bbox1, bbox2):
    x = bbox1.xmin + 0.1
    y = bbox1.ymin
    dx = bbox2.xmax - x - 0.1
    dy = bbox2.ymax - y
    arrow = Arrow(dx=dx, dy=dy, arrowprops=arrowprops)
    return arrow.draw(ax, x, y)

def right_arrow(ax, bbox1, bbox2):
    x = bbox1.xmax - 0.1
    y = bbox1.ymin
    dx = bbox2.xmin - x + 0.1
    dy = bbox2.ymax - y
    arrow = Arrow(dx=dx, dy=dy, arrowprops=arrowprops)
    return arrow.draw(ax, x, y)
```

```
from diagram import diagram, adjust, Bbox

width, height, x, y = [4.94, 2.16, -1.03, 1.91]
ax = diagram(width, height)

dx = 0.6
dy = 0.55

bboxes = []
bboxes.append(frames[4].draw(ax, x+6*dx, y))

bboxes.append(frames[3].draw(ax, x+4*dx, y-dy))
bboxes.append(frames[2].draw(ax, x+8*dx, y-dy))

bboxes.append(frames[2].draw(ax, x+3*dx, y-2*dy))
bboxes.append(frames[1].draw(ax, x+5*dx, y-2*dy))
bboxes.append(frames[1].draw(ax, x+7*dx, y-2*dy))
bboxes.append(frames[0].draw(ax, x+9*dx, y-2*dy))

bboxes.append(frames[1].draw(ax, x+2*dx, y-3*dy))
bboxes.append(frames[0].draw(ax, x+4*dx, y-3*dy))

left_arrow(ax, bboxes[0], bboxes[1])
left_arrow(ax, bboxes[1], bboxes[3])
left_arrow(ax, bboxes[3], bboxes[7])
left_arrow(ax, bboxes[2], bboxes[5])

right_arrow(ax, bboxes[0], bboxes[2])
right_arrow(ax, bboxes[1], bboxes[4])
right_arrow(ax, bboxes[2], bboxes[6])
right_arrow(ax, bboxes[3], bboxes[8])

bbox = Bbox.union(bboxes)
# adjust(x, y, bbox)
```

Un call graph muestra un conjunto de función frames, con líneas que conectan cada frame con los frames de las funciones que llama. En la parte superior del graph, `fibonacci` con `n=4` llama a `fibonacci` con `n=3` y `n=2`. A su vez, `fibonacci` con `n=3` llama a `fibonacci` con `n=2` y `n=1`. Y así sucesivamente.

Cuenta cuántas veces se llama a `fibonacci(0)` y `fibonacci(1)`. Esta es una solución ineficiente al problema, y empeora a medida que el argumento crece.

Una solución es llevar un registro de los valores que ya se han calculado almacenándolos en un diccionario. Un valor calculado previamente que se almacena para usarlo más tarde se llama **memo**. Aquí tienes una versión “memoized” de `fibonacci`:

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

def fibonacci_memo(n):
    if n in known:
        return known[n]

    res = fibonacci_memo(n-1) + fibonacci_memo(n-2)
    known[n] = res
    return res
```

`known` es un diccionario que lleva el registro de los números de Fibonacci que ya conocemos. Empieza con dos elementos: `0` mapea a `0` y `1` mapea a `1`.

Cada vez que se llama a `fibonacci_memo`, comprueba `known`. Si el resultado ya está ahí, puede devolverlo inmediatamente. De lo contrario, tiene que calcular el nuevo valor, añadirlo al diccionario y devolverlo.

Comparando las dos funciones, `fibonacci(40)` tarda unos 30 segundos en ejecutarse. `fibonacci_memo(40)` tarda unos 30 microsegundos, así que es un millón de veces más rápida. En el notebook de este capítulo, verás de dónde vienen estas mediciones.

```
# %time fibonacci(40)
```

```
%time fibonacci_memo(40)
```

10.9. Depuración

A medida que trabajas con datasets más grandes, puede volverse inmanejable depurar imprimiendo y comprobando la salida a mano. Aquí tienes algunas sugerencias para depurar datasets grandes:

1. Reduce el tamaño de la entrada: Si es posible, reduce el tamaño del dataset. Por ejemplo, si el programa lee un archivo de texto, empieza solo con las primeras 10 líneas, o con el ejemplo más pequeño que puedas encontrar. Puedes editar los archivos directamente o, mejor, modificar el programa para que lea solo las primeras `n` líneas.

   Si hay un error, puedes reducir `n` al valor más pequeño donde ocurre el error. A medida que encuentres y corrijas errores, puedes aumentar `n` gradualmente.

2. Comprueba resúmenes y tipos: En lugar de imprimir y comprobar el dataset completo, considera imprimir resúmenes de los datos; por ejemplo, el número de elementos en un diccionario o el total de una lista de números.

   Una causa común de errores en tiempo de ejecución es un valor que no tiene el tipo correcto. Para depurar de este tipo de error, a menudo basta con imprimir el tipo de un valor.

3. Escribe autocomprobaciones: A veces puedes escribir código para comprobar errores automáticamente. Por ejemplo, si estás calculando la media de una lista de números, podrías comprobar que el resultado no sea mayor que el elemento más grande de la lista ni menor que el más pequeño. Esto se llama una "sanity check" porque detecta resultados que son "insane".

   Otro tipo de comprobación compara los resultados de dos computaciones diferentes para ver si son consistentes. Esto se llama una "consistency check".

4. Formatea la salida: Formatear la salida de depuración puede facilitar detectar un error. Vimos un ejemplo en Capítulo 6. Otra herramienta que puede resultarte útil es el módulo `pprint`, que proporciona una función `pprint` que muestra tipos integrados en un formato más legible para humanos (`pprint` significa "pretty print").

   De nuevo, el tiempo que dediques a construir código de apoyo puede reducir el tiempo que dedicas a depurar.

10.10. Glosario

diccionario: Un objeto que contiene pares clave-valor, también llamados elementos.

elemento: En un diccionario, otro nombre para un par clave-valor.

clave: Un objeto que aparece en un diccionario como la primera parte de un par clave-valor.

valor: Un objeto que aparece en un diccionario como la segunda parte de un par clave-valor. Esto es más específico que nuestro uso anterior de la palabra "valor".

mapeo: Una relación en la que cada elemento de un conjunto corresponde a un elemento de otro conjunto.

hash table: Una colección de pares clave-valor organizada para que podamos buscar una clave y encontrar su valor de forma eficiente.

hasheable: Tipos inmutables como enteros, floats y strings son hasheable. Tipos mutables como listas y diccionarios no lo son.

función hash: Una función que toma un objeto y calcula un entero que se usa para localizar una clave en una hash table.

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

filtrado: Recorrer una secuencia y seleccionar u omitir elementos.

call graph: Un diagrama que muestra cada frame creado durante la ejecución de un programa, con una flecha desde cada código que llama hacia cada callee.

memo: Un valor calculado almacenado para evitar computación futura innecesaria.

10.11. Ejercicios

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

10.11.1. Pregunta a un asistente virtual

En este capítulo, dije que las claves de un diccionario tienen que ser hasheabless y di una explicación breve. Si quieres más detalles, pregunta a un asistente virtual: "¿Por qué las claves de los diccionarios de Python tienen que ser hasheabless?"

En una sección anterior, almacenamos una lista de palabras como claves en un diccionario para poder usar una versión eficiente del operador `in`. Podríamos haber hecho lo mismo usando un `set`, que es otro tipo de dato integrado. Pregunta a un asistente virtual: "How do I make a Python set from a lista of strings and check whether a string is an elemento of the set?"

10.11.2. Ejercicio

Los diccionarios tienen un método llamado `get` que toma una clave y un valor por defecto. Si la clave aparece en el diccionario, `get` devuelve el valor correspondiente; de lo contrario devuelve el

valor por defecto. Por ejemplo, aquí tienes un diccionario que mapea las letras de un string al número de veces que aparecen.

```
counter = value_counts('brontosaurus')
```

Si buscamos una letra que aparece en la palabra, `get` devuelve el número de veces que aparece.

```
counter.get('b', 0)
```

Si buscamos una letra que no aparece, obtenemos el valor por defecto, `0`.

```
counter.get('c', 0)
```

Usa `get` para escribir una versión más concisa de `value_counts`. Deberías poder eliminar la sentencia `if`.

10.11.3. Ejercicio

¿Cuál es la palabra más larga que se te ocurre donde cada letra aparece solo una vez? Veamos si podemos encontrar una más larga que `unpredictably`.

Escribe una función llamada `has_duplicates` que tome una secuencia, como una lista o string, como parameter y devuelva `True` si hay algún elemento que aparece en la secuencia más de una vez.

Para empezar, aquí tienes un esquema de la función con doctests.

```
def has_duplicates(t):
    """Check whether any element in a sequence appears more than once.

    >>> has_duplicates('banana')
    True
    >>> has_duplicates('ambidextrously')
    False
    >>> has_duplicates([1, 2, 2])
    True
    >>> has_duplicates([1, 2, 3])
    False
    """
    return None
```

Puedes usar `doctest` para probar tu función.

```
from doctest import run_docstring_examples

def run_doctests(func):
    run_docstring_examples(func, globals(), name=func.__name__)

run_doctests(has_duplicates)
```

Puedes usar este bucle para encontrar las palabras más largas sin letras repetidas.

```
no_repeats = []

for word in word_list:
    if len(word) > 12 and not has_duplicates(word):
        no_repeats.append(word)

no_repeats
```

10.11.4. Ejercicio

Escribe una función llamada `find_repeats` que tome un diccionario que mapea cada clave a un contador, como el resultado de `value_counts`. Debe recorrer el diccionario y devolver una lista de claves que tengan recuentos mayores que `1`. Puedes usar el siguiente esquema para empezar.

```
def find_repeats(counter):
    """Makes a list of keys with values greater than 1.

    counter: dictionary that maps from keys to counts

    returns: list of keys
    """
    return []
```

Puedes usar los siguientes ejemplos para probar tu código. Primero, haremos un diccionario que mapea letras a recuentos.

```
counter1 = value_counts('banana')
counter1
```

El resultado de `find_repeats` debería ser `['a', 'n']`.

```
repeats = find_repeats(counter1)
repeats
```

Aquí tienes otro ejemplo que empieza con una lista de números. El resultado debería ser `[1, 2]`.

```
counter1 = value_counts([1, 2, 3, 2, 1])
repeats = find_repeats(counter1)
repeats
```

10.11.5. Ejercicio

Supongamos que ejecutas `value_counts` con dos palabras diferentes y guardas los resultados en dos diccionarios.

```
counter1 = value_counts('brontosaurus')
counter2 = value_counts('apatosaurus')
```

Cada diccionario mapea un conjunto de letras al número de veces que aparecen. Escribe una función llamada `add_counters` que tome dos diccionarios como estos y devuelva un nuevo diccionario que contenga todas las letras y el número total de veces que aparecen en cualquiera de las dos palabras.

Hay muchas formas de resolver este problema. Cuando tengas una solución que funcione, considera pedir a un asistente virtual soluciones diferentes.

10.11.6. Ejercicio

Una palabra es “interlocking” si podemos dividirla en dos palabras tomando letras alternas. Por ejemplo, “schooled” es una palabra interlocking porque puede dividirse en “shoe” y “cold”.

Para seleccionar letras alternas de un string, puedes usar un operador de porción con tres componentes que indican dónde empezar, dónde detenerse y el “step size” entre las letras.

En el siguiente porción, el primer componente es `0`, así que empezamos con la primera letra. El segundo componente es `None`, lo que significa que debemos llegar hasta el final del string. Y el tercer componente es `2`, así que hay dos pasos entre las letras que seleccionamos.

```
word = 'schooled'
first = word[0:None:2]
first
```

En lugar de proporcionar `None` como segundo componente, podemos obtener el mismo efecto omitiéndolo por completo. Por ejemplo, el siguiente porción selecciona letras alternas, empezando por la segunda letra.

```
second = word[1::2]
second
```

Escribe una función llamada `is_interlocking` que tome una palabra como argumento y devuelva `True` si puede dividirse en dos palabras interlocking.

Puedes usar el siguiente bucle para encontrar las palabras interlocking en la palabra lista.

```
for word in word_list:
    if len(word) >= 8 and is_interlocking(word):
        first = word[0::2]
        second = word[1::2]
        print(word, first, second)
```

11. Tuplas

Este capítulo presenta un tipo integrado más, la tupla, y luego muestra cómo listas, diccionarios y tuplas trabajan juntos. También presenta la asignación de tuplas y una característica útil para funciones con listas de argumentos de longitud variable: los operadores de empaquetado y desempaquetado.

En los ejercicios, usaremos tuplas, junto con listas y diccionarios, para resolver más puzles de palabras e implementar algoritmos eficientes.

Una nota: hay dos maneras de pronunciar “tupla”. Algunas personas dicen “tuh-ple”, que rima con “supple”. Pero en el contexto de la programación, la mayoría dice “too-ple”, que rima con “quadruple”.

11.1. Las tuplas son como listas

Una tupla es una secuencia de valores. Los valores pueden ser de cualquier tipo, y se indexan con enteros, así que las tuplas se parecen mucho a las listas. La diferencia importante es que las tuplas son inmutables.

Para crear una tupla, puedes escribir una lista de valores separados por comas.

```
t = 'l', 'u', 'p', 'i', 'n'
type(t)
```

Aunque no es necesario, es común encerrar las tuplas entre paréntesis.

```
t = ('l', 'u', 'p', 'i', 'n')
type(t)
```

Para crear una tupla con un solo elemento, tienes que incluir una coma final.

```
t1 = 'p',
type(t1)
```

Un único valor entre paréntesis no es una tupla.

```
t2 = ('p')
type(t2)
```

Otra forma de crear una tupla es la función integrada `tuple`. Sin argumentos, crea una tupla vacío.

```
t = tuple()
t
```

Si el argumento es una secuencia (string, lista o tupla), el resultado es una tupla con los elementos de la secuencia.

```
t = tuple('lupin')
t
```

Como `tuple` es el nombre de una función integrada, deberías evitar usarlo como nombre de variable.

La mayoría de los operadores de lista también funcionan con tuplas. Por ejemplo, el operador de corchetes indexa un elemento.

```
t[0]
```

Y el operador de porción selecciona un rango de elementos.

```
t[1:3]
```

El operador `+` concatena tuplas.

```
tuple('lup') + ('i', 'n')
```

Y el operador `*` duplica una tupla un número dado de veces.

```
tuple('spam') * 2
```

La función `sorted` funciona con tuplas – pero el resultado es una lista, no una tupla.

```
sorted(t)
```

La función `reversed` también funciona con tuplas.

```
reversed(t)
```

El resultado es un objeto `reversed`, que podemos convertir en una lista o en una tupla.

```
tuple(reversed(t))
```

Según los ejemplos hasta ahora, podría parecer que las tuplas son iguales que las listas.

11.2. Pero las tuplas son inmutables

Si intentas modificar una tupla con el operador de corchetes, obtienes un `TypeError`.

```
t[0] = 'L'
```

Y las tuplas no tienen ninguno de los métodos que modifican listas, como `append` y `remove`.

```
t.remove('l')
```

Recuerda que un “atributo” es una variable o método asociado con un objeto – este mensaje de error significa que las tuplas no tienen un método llamado `remove`.

Como las tuplas son inmutables, son hasheables, lo que significa que pueden usarse como claves en un diccionario. Por ejemplo, el siguiente diccionario contiene dos tuplas como claves que se asocian con enteros.

```
d = {}
d[1, 2] = 3
d[3, 4] = 7
```

Podemos buscar una tupla en un diccionario así:

```
d[1, 2]
```

O, si tenemos una variable que se refiere a una tupla, podemos usarla como clave.

```
t = (3, 4)
d[t]
```

Las tuplas también pueden aparecer como valores en un diccionario.

```
t = tuple('abc')
d = {'key': t}
d
```

11.3. Asignación de tupla

Puedes poner una tupla de variables en el lado izquierdo de una asignación, y una tupla de valores en el lado derecho.

```
a, b = 1, 2
```

Los valores se asignan a las variables de izquierda a derecha – en este ejemplo, `a` recibe el valor `1` y `b` recibe el valor `2`. Podemos mostrar los resultados así:

```
a, b
```

De forma más general, si el lado izquierdo de una asignación es una tupla, el lado derecho puede ser cualquier tipo de secuencia – string, lista o tupla. Por ejemplo, para dividir una dirección de email en un nombre de usuario y un dominio, podrías escribir:

```
email = 'monty@python.org'
username, domain = email.split('@')
```

El valor de retorno de `split` es una lista con dos elementos – el primer elemento se asigna a `username`, el segundo a `domain`.

```
username, domain
```

El número de variables a la izquierda y el número de valores a la derecha tienen que ser iguales – de lo contrario obtienes un `ValueError`.

```
a, b = 1, 2, 3
```

La asignación de tupla es útil si quieres intercambiar los valores de dos variables. Con asignaciones convencionales, tienes que usar una variable temporal, así:

```
temp = a
a = b
b = temp
```

Eso funciona, pero con la asignación de tupla podemos hacer lo mismo sin una variable temporal.

```
a, b = b, a
```

Esto funciona porque todas las expresiones del lado derecho se evalúan antes de hacer cualquiera de las asignaciones.

También podemos usar asignación de tupla en una sentencia `for`. Por ejemplo, para hacer un bucle por los elementos de un diccionario, podemos usar el método `items`.

```
d = {'one': 1, 'two': 2}

for item in d.items():
    key, value = item
    print(key, '->', value)
```

Cada vez que pasa por el bucle, `item` se asigna a una tupla que contiene una clave y el valor correspondiente.

Podemos escribir este bucle de forma más concisa, así:

```
for key, value in d.items():
    print(key, '->', value)
```

Cada vez que pasa por el bucle, una clave y el valor correspondiente se asignan directamente a `key` y `value`.

11.4. Tuplas como valores de retorno

Estrictamente hablando, una función solo puede devolver un valor, pero si el valor es una tupla, el efecto es el mismo que devolver múltiples valores. Por ejemplo, si quieres dividir dos enteros y calcular el cociente y el resto, es ineficiente calcular `x//y` y luego `x%y`. Es mejor calcular ambos al mismo tiempo.

La función integrada `divmod` toma dos argumentos y devuelve una tupla de dos valores, el cociente y el resto.

```
divmod(7, 3)
```

Podemos usar asignación de tupla para guardar los elementos dla tupla en dos variables.

```
quotient, remainder = divmod(7, 3)
quotient
```

```
remainder
```

Aquí tienes un ejemplo de una función que devuelve una tupla.

```
def min_max(t):
    return min(t), max(t)
```

`max` y `min` son funciones integradas que encuentran los elementos mayor y menor de una secuencia. `min_max` calcula ambos y devuelve una tupla de dos valores.

```
min_max([2, 4, 1, 3])
```

Podemos asignar los resultados a variables así:

```
low, high = min_max([2, 4, 1, 3])
low, high
```

11.5. Empaquetado de argumentos

Las funciones pueden tomar un número variable de argumentos. Un nombre de parámetro que empieza con el operador `*` **empaqueta** argumentos en una tupla. Por ejemplo, la siguiente función toma cualquier número de argumentos y calcula su media aritmética – es decir, su suma dividida por el número de argumentos.

```
def mean(*args):
    return sum(args) / len(args)
```

El parámetro puede tener cualquier nombre que quieras, pero `args` es convencional. Podemos llamar a la función así:

```
mean(1, 2, 3)
```

Si tienes una secuencia de valores y quieres pasarlos a una función como múltiples argumentos, puedes usar el operador `*` para **desempaquetar** la tupla. Por ejemplo, `divmod` toma exactamente dos argumentos – si pasas una tupla como parámetro, obtienes un error.

```
t = (7, 3)
divmod(t)
```

Aunque la tupla contiene dos elementos, cuenta como un único argumento. Pero si desempaquetas la tupla, se trata como dos argumentos.

```
divmod(*t)
```

Empaquetado y desempaquetado pueden ser útiles si quieres adaptar el comportamiento de una función existente. Por ejemplo, esta función toma cualquier número de argumentos, elimina el menor y el mayor, y calcula la media del resto.

```
def trimmed_mean(*args):
    low, high = min_max(args)
    trimmed = list(args)
    trimmed.remove(low)
    trimmed.remove(high)
    return mean(*trimmed)
```

Primero usa `min_max` para encontrar los elementos menor y mayor. Luego convierte `args` en una lista para poder usar el método `remove`. Finalmente desempaqueta la lista para que los elementos se pasen a `mean` como argumentos separados, en lugar de como una única lista.

Aquí tienes un ejemplo que muestra el efecto.

```
mean(1, 2, 3, 10)
```

```
trimmed_mean(1, 2, 3, 10)
```

Este tipo de media "recortada" se usa en algunos deportes con puntuación subjetiva – como saltos y gimnasia – para reducir el efecto de un juez cuya puntuación se desvía de las demás.

11.6. Zip

Las tuplas son útiles para hacer un bucle por los elementos de dos secuencias y realizar operaciones sobre elementos correspondientes. Por ejemplo, supongamos que dos equipos juegan

una serie de siete partidos, y registramos sus puntuaciones en dos listas, una para cada equipo.

```
scores1 = [1, 2, 4, 5, 1, 5, 2]
scores2 = [5, 5, 2, 2, 5, 2, 3]
```

Veamos cuántos partidos ganó cada equipo. Usaremos `zip`, que es una función integrada que toma dos o más secuencias y devuelve un **zip objeto**, llamado así porque empareja los elementos de las secuencias como los dientes de una cremallera.

```
zip(scores1, scores2)
```

Podemos usar el zip objeto para hacer un bucle por los valores de las secuencias por pares.

```
for pair in zip(scores1, scores2):
     print(pair)
```

Cada vez que pasa por el bucle, `pair` recibe una tupla de puntuaciones. Así que podemos asignar las puntuaciones a variables y contar las victorias del primer equipo, así:

```
wins = 0
for team1, team2 in zip(scores1, scores2):
    if team1 > team2:
        wins += 1

wins
```

Tristemente, el primer equipo ganó solo tres partidos y perdió la serie.

Si tienes dos listas y quieres una lista de pares, puedes usar `zip` y `list`.

```
t = list(zip(scores1, scores2))
t
```

El resultado es una lista de tuplas, así que podemos obtener el resultado del último partido así:

```
t[-1]
```

Si tienes una lista de claves y una lista de valores, puedes usar `zip` y `dict` para crear un diccionario. Por ejemplo, así podemos crear un diccionario que asocia cada letra con su posición en el alfabeto.

```
letters = 'abcdefghijklmnopqrstuvwxyz'
numbers = range(len(letters))
letter_map = dict(zip(letters, numbers))
```

Ahora podemos buscar una letra y obtener su índice en el alfabeto.

```
letter_map['a'], letter_map['z']
```

En este mapeo, el índice de `'a'` es `0` y el índice de `'z'` es `25`.

Si necesitas hacer un bucle por los elementos de una secuencia y sus índices, puedes usar la función integrada `enumerate`.

```
enumerate('abc')
```

El resultado es un **enumerate objeto** que hace un bucle por una secuencia de pares, donde cada par contiene un índice (empezando en 0) y un elemento de la secuencia dada.

```
for index, element in enumerate('abc'):
    print(index, element)
```

11.7. Comparación y ordenación

Los operadores relacionales funcionan con tuplas y otras secuencias. Por ejemplo, si usas el operador `<` con tuplas, empieza comparando el primer elemento de cada secuencia. Si son iguales, continúa con el siguiente par de elementos, y así sucesivamente, hasta que encuentra un par que difiere.

```
(0, 1, 2) < (0, 3, 4)
```

Los elementos posteriores no se tienen en cuenta – aunque sean realmente grandes.

```
(0, 1, 2000000) < (0, 3, 4)
```

Esta forma de comparar tuplas es útil para ordenar una lista de tuplas, o para encontrar el mínimo o el máximo. Como ejemplo, encontremos la letra más común en una palabra. En el capítulo anterior, escribimos `value_counts`, que toma un string y devuelve un diccionario que asocia cada letra con el número de veces que aparece.

```
def value_counts(string):
    counter = {}
    for letter in string:
        if letter not in counter:
            counter[letter] = 1
        else:
            counter[letter] += 1
    return counter
```

Aquí está el resultado para el string `'banana'`.

```
counter = value_counts('banana')
counter
```

Con solo tres elementos, podemos ver fácilmente que la letra más frecuente es `'a'`, que aparece tres veces. Pero si hubiera más elementos, sería útil ordenarlos automáticamente.

Podemos obtener los elementos de `counter` así.

```
items = counter.items()
items
```

El resultado es un objeto `dict_items` que se comporta como una lista de tuplas, así que podemos ordenarlo así.

```
sorted(items)
```

El comportamiento por defecto es usar el primer elemento de cada tupla para ordenar la lista, y usar el segundo elemento para desempatar.

Sin embargo, para encontrar los elementos con los conteos más altos, queremos usar el segundo elemento para ordenar la lista. Podemos hacerlo escribiendo una función que toma una tupla y devuelve el segundo elemento.

```
def second_element(t):
    return t[1]
```

Luego podemos pasar esa función a `sorted` como un argumento opcional llamado `key`, que indica que esta función debe usarse para calcular la **sort clave** de cada elemento.

```
sorted_items = sorted(items, key=second_element)
sorted_items
```

La sort clave determina el orden de los elementos en la lista. La letra con el conteo más bajo aparece primero, y la letra con el conteo más alto aparece al final. Así podemos encontrar la letra más común.

```
sorted_items[-1]
```

Si solo queremos el máximo, no tenemos que ordenar la lista. Podemos usar `max`, que también toma `key` como argumento opcional.

```
max(items, key=second_element)
```

Para encontrar la letra con el conteo más bajo, podríamos usar `min` de la misma manera.

11.8. Invertir un diccionario

Supongamos que quieres invertir un diccionario para poder buscar un valor y obtener la clave correspondiente. Por ejemplo, si tienes un contador de palabras que asocia cada palabra con el número de veces que aparece, podrías crear un diccionario que asocia enteros con las palabras que aparecen ese número de veces.

Pero hay un problema – las claves de un diccionario tienen que ser únicas, pero los valores no. Por ejemplo, en un contador de palabras, podría haber muchas palabras con el mismo conteo.

Así que una forma de invertir un diccionario es crear un nuevo diccionario donde los valores son listas de claves del original. Como ejemplo, contemos las letras en `parrot`.

```
d =  value_counts('parrot')
d
```

Si invertimos este diccionario, el resultado debería ser `{1: ['p', 'a', 'o', 't'], 2: ['r']}`, lo que indica que las letras que aparecen una vez son `'p'`, `'a'`, `'o'` y `'t'`, y la letra que aparece dos veces es `'r'`.

La siguiente función toma un diccionario y devuelve su inverso como un nuevo diccionario.

```
def invert_dict(d):
    new = {}
    for key, value in d.items():
        if value not in new:
            new[value] = [key]
        else:
            new[value].append(key)
    return new
```

La sentencia `for` hace un bucle por las claves y los valores de `d`. Si el valor todavía no está en el nuevo diccionario, se añade y se asocia con una lista que contiene un único elemento. De lo contrario, se añade a la lista existente.

Podemos probarlo así:

```
invert_dict(d)
```

Y obtenemos el resultado que esperábamos.

Este es el primer ejemplo que hemos visto donde los valores en el diccionario son listas. ¡Veremos más!

11.9. Depuración

Listas, diccionarios y tuplas son **estructuras de datos**. En este capítulo estamos empezando a ver estructuras de datos compuestas, como listas de tuplas, o diccionarios que contienen tuplas como

claves y listas como valores. Las estructuras de datos compuestas son útiles, pero son propensas a errores cuando una estructura de datos tiene el tipo, tamaño o estructura equivocados. Por ejemplo, si una función espera una lista de enteros y le das un entero sin más (no dentro de una lista), probablemente no funcionará.

Para ayudar a depurar este tipo de errores, escribí un módulo llamado `structshape` que proporciona una función, también llamada `structshape`, que toma cualquier tipo de estructura de datos como argumento y devuelve un string que resume su estructura. Puedes descargarlo desde https://raw.githubusercontent.com/AllenDowney/ThinkPython/v3/structshape.py.

```
download('https://raw.githubusercontent.com/AllenDowney/ThinkPython/v3/structshape.
```

Podemos importarlo así.

```
from structshape import structshape
```

Aquí tienes un ejemplo con una lista simple.

```
t = [1, 2, 3]
structshape(t)
```

Aquí tienes una lista de listas.

```
t2 = [[1,2], [3,4], [5,6]]
structshape(t2)
```

Si los elementos de la lista no son del mismo tipo, `structshape` los agrupa por tipo.

```
t3 = [1, 2, 3, 4.0, '5', '6', [7], [8], 9]
structshape(t3)
```

Aquí tienes una lista de tuplas.

```
s = 'abc'
lt = list(zip(t, s))
structshape(lt)
```

Y aquí tienes un diccionario con tres elementos que asocian enteros con strings.

```
d = dict(lt)
structshape(d)
```

Si tienes problemas para seguir la pista de tus estructuras de datos, `structshape` puede ayudar.

11.10. Glosario

pack: Reunir múltiples argumentos en una tupla.

unpack: Tratar una tupla (u otra secuencia) como múltiples argumentos.

zip objeto: El resultado de llamar a la función integrada `zip`; se puede usar para hacer un bucle por una secuencia de tuplas.

enumerate objeto: El resultado de llamar a la función integrada `enumerate`; se puede usar para hacer un bucle por una secuencia de tuplas.

sort clave: Un valor, o una función que calcula un valor, usado para ordenar los elementos de una colección.

estructura de datos: Una colección de valores, organizada para realizar ciertas operaciones de manera eficiente.

11.11. Ejercicios

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

11.11.1. Pregunta a un asistente virtual

Los ejercicios de este capítulo pueden ser más difíciles que los de capítulos anteriores, así que te animo a pedir ayuda a un asistente virtual. Cuando planteas preguntas más difíciles, puede que las

respuestas no sean correctas al primer intento, así que esta es una oportunidad para practicar cómo crear buenos prompts y hacer buenos refinamientos.

Una estrategia que podrías considerar es dividir un problema grande en piezas que puedan resolverse con funciones simples. Pide al asistente virtual que escriba las funciones y las pruebe. Luego, cuando funcionen, pide una solución al problema original.

Para algunos de los ejercicios siguientes, sugiero qué estructuras de datos y algoritmos usar. Puede que estas sugerencias te resulten útiles cuando trabajes en los problemas, pero también son buenos prompts para pasárselos a un asistente virtual.

11.11.2. Ejercicio

En este capítulo dije que las tuplas pueden usarse como claves en diccionarios porque son hasheables, y son hasheables porque son inmutables. Pero eso no siempre es cierto.

Si una tupla contiene un valor mutable, como una lista o un diccionario, la tupla ya no es hasheable porque contiene elementos que no son hasheablesss. Como ejemplo, aquí tienes una tupla que contiene dos listas de enteros.

```
list0 = [1, 2, 3]
list1 = [4, 5]

t = (list0, list1)
t
```

Escribe una línea de código que añada el valor `6` al final de la segunda lista en `t`. Si muestras `t`, el resultado debería ser `([1, 2, 3], [4, 5, 6])`.

Intenta crear un diccionario que asocie `t` con un string, y confirma que obtienes un `TypeError`.

Para saber más sobre este tema, pregunta a un asistente virtual: «Are Python tuplas always hasheable?

11.11.3. Ejercicio

En este capítulo hicimos un diccionario que asocia cada letra con su índice en el alfabeto.

```
letters = 'abcdefghijklmnopqrstuvwxyz'
numbers = range(len(letters))
letter_map = dict(zip(letters, numbers))
```

Por ejemplo, el índice de `'a'` es `0`.

```
letter_map['a']
```

Para ir en la otra dirección, podemos usar indexación de lista. Por ejemplo, la letra en el índice `1` es `'b'`.

```
letters[1]
```

Podemos usar `letter_map` y `letters` para codificar y decodificar palabras usando un cifrado César.

Un cifrado César es una forma débil de cifrado que consiste en desplazar cada letra un número fijo de posiciones en el alfabeto, volviendo al principio si es necesario. Por ejemplo, `'a'` desplazada 2 posiciones es `'c'` y `'z'` desplazada 1 posición es `'a'`.

Escribe una función llamada `shift_word` que tome como parámetros un string y un entero, y devuelva un nuevo string que contiene las letras del string desplazadas el número de posiciones dado.

Para probar tu función, confirma que “cheer” desplazada 7 posiciones es “jolly” y “melon” desplazada 16 posiciones es “cubed”.

Pistas: usa el operador módulo para volver de `'z'` a `'a'`. Haz un bucle por las letras de la palabra, desplaza cada una y añade el resultado a una lista de letras. Luego usa `join` para concatenar las letras en un string.

Puedes usar este esquema para empezar.

```
def shift_word(word, n):
    """Shift the letters of `word` by `n` places.

    >>> shift_word('cheer', 7)
    'jolly'
    >>> shift_word('melon', 16)
    'cubed'
    """
    return None
```

```
shift_word('cheer', 7)
```

```
shift_word('melon', 16)
```

Puedes usar `doctest` para probar tu función.

```
from doctest import run_docstring_examples

def run_doctests(func):
    run_docstring_examples(func, globals(), name=func.__name__)

run_doctests(shift_word)
```

11.11.4. Ejercicio

Escribe una función llamada `most_frequent_letters` que tome un string e imprima las letras en orden decreciente de frecuencia.

Para obtener los elementos en orden decreciente, puedes usar `reversed` junto con `sorted` o puedes pasar `reverse=True` como keyword parameter a `sorted`.

Puedes usar este esquema de la función para empezar.

```
def most_frequent_letters(string):
    return None
```

Y este ejemplo para probar tu función.

```
most_frequent_letters('brontosaurus')
```

Cuando tu función funcione, puedes usar el siguiente código para imprimir las letras más comunes en *Dracula*, que podemos descargar de Project Gutenberg.

```
download('https://www.gutenberg.org/cache/epub/345/pg345.txt');
```

```
string = open('pg345.txt').read()
most_frequent_letters(string)
```

Según *Codes and Secret Writing* de Zim, la secuencia de letras en orden decreciente de frecuencia en inglés empieza con "ETAONRISH". ¿Cómo se compara esta secuencia con los resultados de *Dracula*?

11.11.5. Ejercicio

En un ejercicio anterior, probamos si dos strings son anagramas ordenando las letras de ambas palabras y comprobando si las letras ordenadas son iguales. Ahora hagamos el problema un poco más desafiante.

Escribiremos un programa que toma una lista de palabras e imprime todos los conjuntos de palabras que son anagramas. Aquí tienes un ejemplo de cómo podría verse la salida:

```
['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
['retainers', 'ternaries']
['generating', 'greatening']
['resmelts', 'smelters', 'termless']
```

Pista: para cada palabra de la lista de palabras, ordena las letras y vuelve a unirlas en un string. Crea un diccionario que asocie este string ordenado con una lista de palabras que son anagramas de él.

Las siguientes celdas descargan `words.txt` y leen las palabras en una lista.

```
download('https://raw.githubusercontent.com/AllenDowney/ThinkPython/v3/words.txt');
```

```
word_list = open('words.txt').read().split()
```

Aquí está la función `sort_word` que hemos usado antes.

```
def sort_word(word):
    return ''.join(sorted(word))
```

Para encontrar la lista más larga de anagramas, puedes usar la siguiente función, que toma un par clave-valor donde la clave es un string y el valor es una lista de palabras. Devuelve la longitud de la lista.

```
def value_length(pair):
    key, value = pair
    return len(value)
```

Podemos usar esta función como sort clave para encontrar las listas más largas de anagramas.

```
anagram_items = sorted(anagram_dict.items(), key=value_length)
for key, value in anagram_items[-10:]:
    print(value)
```

Si quieres saber cuáles son las palabras más largas que tienen anagramas, puedes usar el siguiente bucle para imprimir algunas.

```
longest = 7

for key, value in anagram_items:
    if len(value) > 1:
        word_len = len(value[0])
        if word_len > longest:
            longest = word_len
            print(value)
```

11.11.6. Ejercicio

Escribe una función llamada `word_distance` que tome dos palabras con la misma longitud y devuelva el número de posiciones donde las dos palabras difieren.

Pista: usa `zip` para hacer un bucle por las letras correspondientes de las palabras.

Aquí tienes un esquema de la función con doctests que puedes usar para comprobar tu función.

```
def word_distance(word1, word2):
    """Computes the number of places where two word differ.

    >>> word_distance("hello", "hxllo")
    1
    >>> word_distance("ample", "apply")
    2
    >>> word_distance("kitten", "mutton")
    3
    """
    return None
```

```
from doctest import run_docstring_examples

def run_doctests(func):
    run_docstring_examples(func, globals(), name=func.__name__)

run_doctests(word_distance)
```

11.11.7. Ejercicio

"Metathesis" es la transposición de letras en una palabra. Dos palabras forman una "metathesis pair" si puedes transformar una en la otra intercambiando dos letras, como `converse` y `conserve`. Escribe un programa que encuentre todos los pares de metathesis en la lista de palabras.

Pista: las palabras de un par de metathesis deben ser anagramas entre sí.

Crédito: este ejercicio está inspirado en un ejemplo de http://puzzlers.org.

11.11.8. Ejercicio

Este es un ejercicio extra que no está en el libro. Es más desafiante que los otros ejercicios de este capítulo, así que quizá quieras pedir ayuda a un asistente virtual, o volver a él después de leer algunos capítulos más.

Aquí tienes otro Car Talk Puzzler (http://www.cartalk.com/content/puzzlers):

> ¿Cuál es la palabra inglesa más larga que sigue siendo una palabra inglesa válida mientras le quitas letras una a una?
>
> Ahora, las letras se pueden quitar de cualquiera de los extremos o del medio, pero no puedes reordenar ninguna de las letras. Cada vez que quitas una letra, terminas con otra palabra inglesa. Si haces eso, al final terminarás con una letra, y esa también será una palabra inglesa – una que aparece en el diccionario. Quiero saber cuál es la palabra más larga y cuántas letras tiene.
>
> Te voy a dar un ejemplo pequeño: Sprite. ¿Ok? Empiezas con sprite, le quitas una letra, una del interior de la palabra, quitas la r, y nos queda la palabra spite; luego quitamos la e del final, nos queda spit; quitamos la s, nos queda pit, it e I.

Escribe un programa para encontrar todas las palabras que pueden reducirse de esta manera, y luego encuentra la más larga.

Este ejercicio es un poco más desafiante que la mayoría, así que aquí tienes algunas sugerencias:

1. Quizá quieras escribir una función que tome una palabra y calcule una lista de todas las palabras que pueden formarse quitando una letra. Estas son las "children" de la palabra.
2. Recursivamente, una palabra es reducible si cualquiera de sus children es reducible. Como caso base, puedes considerar que el string vacío es reducible.
3. La lista de palabras que hemos estado usando no contiene palabras de una sola letra. Así que quizá tengas que añadir "I" y "a".
4. Para mejorar el rendimiento de tu programa, quizá quieras memoizar las palabras que se sabe que son reducibles.

Puedes pedir las versiones impresa y ebook de *Think Python 3e* en Bookshop.org y Amazon.

12. Análisis y generación de texto

A estas alturas hemos cubierto las estructuras de datos principales de Python – listas, diccionarios y tuplas – y algunos algoritmos que las usan. En este capítulo, las usaremos para explorar el análisis de texto y la generación de Markov:

- El análisis de texto es una forma de describir las relaciones estadísticas entre las palabras de un documento, como la probabilidad de que una palabra vaya seguida de otra, y
- La generación de Markov es una forma de generar texto nuevo con palabras y frases similares a las del texto original.

Estos algoritmos son similares a partes de un Large Language Model (LLM), que es el componente clave de un chatbot.

Empezaremos contando el número de veces que aparece cada palabra en un libro. Luego veremos pares de palabras y haremos una lista de las palabras que pueden seguir a cada palabra. Haremos una versión simple de un generador de Markov y, como ejercicio, tendrás la oportunidad de hacer una versión más general.

12.1. Palabras únicas

Como primer paso hacia el análisis de texto, leamos un libro – *The Strange Case Of Dr. Jekyll And Mr. Hyde* de Robert Louis Stevenson – y contemos el número de palabras únicas. Las instrucciones para descargar el libro están en el notebook de este capítulo.

La siguiente celda descarga el libro desde Project Gutenberg.

```
download('https://www.gutenberg.org/cache/epub/43/pg43.txt');
```

La versión disponible en Project Gutenberg incluye información sobre el libro al principio e información de licencia al final. Usaremos `clean_file` del Capítulo 8 para eliminar este material y escribir un archivo “limpio” que contiene solo el texto del libro.

```
def is_special_line(line):
    return line.strip().startswith('*** ')
```

```
def clean_file(input_file, output_file):
    reader = open(input_file, encoding='utf-8')
    writer = open(output_file, 'w')

    for line in reader:
        if is_special_line(line):
            break

    for line in reader:
        if is_special_line(line):
            break
        writer.write(line)

    reader.close()
    writer.close()
```

```
filename = 'dr_jekyll.txt'
```

```
clean_file('pg43.txt', filename)
```

Usaremos un bucle `for` para leer líneas del archivo y `split` para dividir las líneas en palabras. Luego, para llevar la cuenta de las palabras únicas, guardaremos cada palabra como una clave en un diccionario.

```
unique_words = {}
for line in open(filename):
    seq = line.split()
    for word in seq:
        unique_words[word] = 1

len(unique_words)
```

La longitud del diccionario es el número de palabras únicas – unas `6000` con esta forma de contar. Pero si las inspeccionamos, veremos que algunas no son palabras válidas.

Por ejemplo, veamos las palabras más largas en `unique_words`. Podemos usar `sorted` para ordenar las palabras, pasando la función `len` como keyword argumento para que las palabras se ordenen por longitud.

```
sorted(unique_words, key=len)[-5:]
```

El índice de porción, `[-5:]`, selecciona los últimos `5` elementos de la lista ordenada, que son las palabras más largas.

La lista incluye algunas palabras legítimamente largas, como “circumscription”, y algunas palabras con guion, como “chocolate-coloured”. Pero algunas de las “palabras” más largas son en realidad dos palabras separadas por una raya. Y otras palabras incluyen puntuación como puntos, signos de exclamación y comillas.

Así que, antes de seguir, ocupémonos de las rayas y otros signos de puntuación.

12.2. Puntuación

Para identificar las palabras del texto, tenemos que resolver dos cuestiones:

- Cuando aparece una raya en una línea, deberíamos reemplazarla por un espacio – así, cuando usemos `split`, las palabras quedarán separadas.
- Después de separar las palabras, podemos usar `strip` para eliminar la puntuación.

Para manejar la primera cuestión, podemos usar la siguiente función, que toma un string, reemplaza las rayas por espacios, divide el string y devuelve la lista resultante.

```
def split_line(line):
    return line.replace('—', ' ').split()
```

Observa que `split_line` solo reemplaza rayas, no guiones. Aquí tienes un ejemplo.

```
split_line('coolness—frightened')
```

Ahora, para eliminar la puntuación del principio y del final de cada palabra, podemos usar `strip`, pero necesitamos una lista de caracteres que se consideran puntuación.

Los caracteres en los strings de Python están en Unicode, que es un estándar internacional usado para representar letras de casi todos los alfabetos, números, símbolos, signos de puntuación y más. El módulo `unicodedata` proporciona una función `category` que podemos usar para saber

qué caracteres son puntuación. Dada una letra, devuelve un string con información sobre la categoría a la que pertenece.

```
import unicodedata

unicodedata.category('A')
```

El string de categoría de `'A'` es `'Lu'` – la `'L'` significa que es una letra y la `'u'` significa que es mayúscula.

El string de categoría de `'.'` es `'Po'` – la `'P'` significa que es puntuación y la `'o'` significa que su subcategoría es "other".

```
unicodedata.category('.')
```

Podemos encontrar los signos de puntuación del libro comprobando los caracteres con categorías que empiezan por `'P'`. El siguiente bucle guarda los signos de puntuación únicos en un diccionario.

```
punc_marks = {}
for line in open(filename):
    for char in line:
        category = unicodedata.category(char)
        if category.startswith('P'):
            punc_marks[char] = 1
```

Para crear una lista de signos de puntuación, podemos unir las claves del diccionario en un string.

```
punctuation = ''.join(punc_marks)
print(punctuation)
```

Ahora que sabemos qué caracteres del libro son puntuación, podemos escribir una función que toma una palabra, elimina la puntuación del principio y del final, y la convierte a minúsculas.

```
def clean_word(word):
    return word.strip(punctuation).lower()
```

Aquí tienes un ejemplo.

```
clean_word('“Behold!”')
```

Como `strip` elimina caracteres del principio y del final, deja intactas las palabras con guion.

```
clean_word('pocket-handkerchief')
```

Ahora aquí tienes un bucle que usa `split_line` y `clean_word` para identificar las palabras únicas del libro.

```
unique_words2 = {}
for line in open(filename):
    for word in split_line(line):
        word = clean_word(word)
        unique_words2[word] = 1

len(unique_words2)
```

Con esta definición más estricta de lo que es una palabra, hay unas 4000 palabras únicas. Y podemos confirmar que la lista de palabras más largas se ha limpiado.

```
sorted(unique_words2, key=len)[-5:]
```

Ahora veamos cuántas veces se usa cada palabra.

12.3. Frecuencias de palabras

El siguiente bucle calcula la frecuencia de cada palabra única.

```
word_counter = {}
for line in open(filename):
    for word in split_line(line):
        word = clean_word(word)
        if word not in word_counter:
            word_counter[word] = 1
        else:
            word_counter[word] += 1
```

La primera vez que vemos una palabra, inicializamos su frecuencia a `1`. Si volvemos a ver la misma palabra más adelante, incrementamos su frecuencia.

Para ver qué palabras aparecen con más frecuencia, podemos usar `items` para obtener los pares clave-valor de `word_counter`, y ordenarlos por el segundo elemento del par, que es la frecuencia. Primero definiremos una función que selecciona el segundo elemento.

```
def second_element(t):
    return t[1]
```

Ahora podemos usar `sorted` con dos keyword argumentos:

- `key=second_element` significa que los elementos se ordenarán según las frecuencias de las palabras.
- `reverse=True` significa que los elementos se ordenarán en orden inverso, con las palabras más frecuentes primero.

```
items = sorted(word_counter.items(), key=second_element, reverse=True)
```

Aquí están las cinco palabras más frecuentes.

```
for word, freq in items[:5]:
    print(freq, word, sep='\t')
```

En la siguiente sección, encapsularemos este bucle en una función. Y la usaremos para demostrar una nueva característica – parámetros opcionales.

12.4. Parámetros opcionales

Hemos usado funciones integradas que toman parámetros opcionales. Por ejemplo, `round` toma un parámetro opcional llamado `ndigits` que indica cuántos decimales conservar.

```
round(3.141592653589793, ndigits=3)
```

Pero no son solo las funciones integradas – también podemos escribir funciones con parámetros opcionales. Por ejemplo, la siguiente función toma dos parámetros, `word_counter` y `num`.

```
def print_most_common(word_counter, num=5):
    items = sorted(word_counter.items(), key=second_element, reverse=True)

    for word, freq in items[:num]:
        print(freq, word, sep='\t')
```

El segundo parámetro parece una sentencia de asignación, pero no lo es – es un parámetro opcional.

Si llamas a esta función con un argumento, `num` recibe el **valor por defecto**, que es `5`.

```
print_most_common(word_counter)
```

Si llamas a esta función con dos argumentos, el segundo argumento se asigna a `num` en lugar del valor por defecto.

```
print_most_common(word_counter, 3)
```

En ese caso, diríamos que el argumento opcional **override** el valor por defecto.

Si una función tiene parámetros requeridos y opcionales, todos los parámetros requeridos tienen que ir primero, seguidos por los opcionales.

```
def bad_function(n=5, word_counter):
    return None
```

12.5. Resta de diccionarios

Supongamos que queremos revisar la ortografía de un libro – es decir, encontrar una lista de palabras que podrían estar mal escritas. Una forma de hacerlo es encontrar palabras del libro que no aparecen en una lista de palabras válidas. En capítulos anteriores, hemos usado una lista de palabras que se consideran válidas en juegos de palabras como Scrabble. Ahora usaremos esta lista para revisar la ortografía de Robert Louis Stevenson.

Podemos pensar en este problema como una resta de conjuntos – es decir, queremos encontrar todas las palabras de un conjunto (las palabras del libro) que no están en el otro (las palabras de la

lista).

La siguiente celda descarga la lista de palabras.

```
download('https://raw.githubusercontent.com/AllenDowney/ThinkPython/v3/words.txt');
```

Como hemos hecho antes, podemos leer el contenido de `words.txt` y dividirlo en una lista de strings.

```
word_list = open('words.txt').read().split()
```

Luego guardaremos las palabras como claves en un diccionario para poder usar el operador `in` y comprobar rápidamente si una palabra es válida.

```
valid_words = {}
for word in word_list:
    valid_words[word] = 1
```

Ahora, para identificar palabras que aparecen en el libro pero no en la lista de palabras, usaremos `subtract`, que toma dos diccionarios como parámetros y devuelve un nuevo diccionario que contiene todas las claves de uno que no están en el otro.

```
def subtract(d1, d2):
    res = {}
    for key in d1:
        if key not in d2:
            res[key] = d1[key]
    return res
```

Así es como lo usamos.

```
diff = subtract(word_counter, valid_words)
```

Para obtener una muestra de palabras que podrían estar mal escritas, podemos imprimir las palabras más comunes en `diff`.

```
print_most_common(diff)
```

Las “palabras mal escritas” más comunes son en su mayoría nombres y algunas palabras de una sola letra (Mr. Utterson es el amigo y abogado del Dr. Jekyll).

Si seleccionamos palabras que solo aparecen una vez, es más probable que sean errores ortográficos reales. Podemos hacerlo haciendo un bucle por los elementos y creando una lista de palabras con frecuencia `1`.

```
singletons = []
for word, freq in diff.items():
    if freq == 1:
        singletons.append(word)
```

Aquí están los últimos elementos de la lista.

```
singletons[-5:]
```

La mayoría son palabras válidas que no están en la lista de palabras. Pero `'reindue'` parece ser un error ortográfico de `'reinduce'`, así que al menos encontramos un error legítimo.

12.6. Números aleatorios

Como paso hacia la generación de texto de Markov, a continuación elegiremos una secuencia aleatoria de palabras de `word_counter`. Pero primero hablemos de aleatoriedad.

Dados los mismos inputs, la mayoría de los programas de computadora son **determinista**, lo que significa que generan los mismos outputs cada vez. El determinismo suele ser algo bueno, ya que esperamos que el mismo cálculo produzca el mismo resultado. Para algunas aplicaciones, sin embargo, queremos que la computadora sea impredecible. Los juegos son un ejemplo, pero hay más.

Hacer que un programa sea verdaderamente no determinista resulta ser difícil, pero hay formas de simularlo. Una es usar algoritmos que generan números **pseudorandom**. Los números pseudorandom no son verdaderamente aleatorios porque se generan mediante un cálculo determinista, pero con solo mirar los números es prácticamente imposible distinguirlos de números aleatorios.

El módulo `random` proporciona funciones que generan números pseudorandom – a los que de aquí en adelante simplemente llamaré "random". Podemos importarlo así.

```
import random
```

```
# this cell initializes the random number generator so it
# generates the same sequence each time the notebook runs.

random.seed(4)
```

El módulo `random` proporciona una función llamada `choice` que elige un elemento de una lista al azar, con la misma probabilidad de elegir cada elemento.

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

Si llamas a la función otra vez, podrías obtener el mismo elemento de nuevo, o uno diferente.

```
random.choice(t)
```

A largo plazo, esperamos obtener cada elemento aproximadamente el mismo número de veces.

Si usas `choice` con un diccionario, obtienes un `KeyError`.

```
random.choice(word_counter)
```

Para elegir una clave aleatoria, tienes que poner las claves en una lista y luego llamar a `choice`.

```
words = list(word_counter)
random.choice(words)
```

Si generamos una secuencia aleatoria de palabras, no tiene mucho sentido.

```
for i in range(6):
    word = random.choice(words)
    print(word, end=' ')
```

Parte del problema es que no estamos teniendo en cuenta que algunas palabras son más comunes que otras. Los resultados serán mejores si elegimos palabras con distintos "weights", de modo que algunas se elijan más a menudo que otras.

Si usamos los valores de `word_counter` como weights, cada palabra se elige con una probabilidad que depende de su frecuencia.

```
weights = word_counter.values()
```

El módulo `random` proporciona otra función llamada `choices` que toma weights como argumento opcional.

```
random.choices(words, weights=weights)
```

Y toma otro argumento opcional, `k`, que especifica el número de palabras a seleccionar.

```
random_words = random.choices(words, weights=weights, k=6)
random_words
```

El resultado es una lista de strings que podemos unir en algo que se parece más a una oración.

```
' '.join(random_words)
```

Si eliges palabras del libro al azar, obtienes una idea del vocabulario, pero una serie de palabras aleatorias rara vez tiene sentido porque no hay relación entre palabras sucesivas. Por ejemplo, en una oración real esperas que un artículo como "the" vaya seguido de un adjetivo o un sustantivo, y probablemente no de un verbo o adverbio. Así que el siguiente paso es observar estas relaciones entre palabras.

12.7. Bigramas

En lugar de mirar una palabra cada vez, ahora miraremos secuencias de dos palabras, que se llaman **bigramas**. Una secuencia de tres palabras se llama **trigram**, y una secuencia con un número no especificado de palabras se llama **n-gram**.

Escribamos un programa que encuentre todos los bigramas del libro y el número de veces que aparece cada uno. Para guardar los resultados, usaremos un diccionario donde

- Las claves son tuplas de strings que representan bigramas, y
- Los valores son enteros que representan frecuencias.

Llamémoslo `bigram_counter`.

```
bigram_counter = {}
```

La siguiente función toma una lista de dos strings como parámetro. Primero crea una tupla con los dos strings, que puede usarse como clave en un diccionario. Luego añade la clave a `bigram_counter`, si no existe, o incrementa la frecuencia si existe.

```
def count_bigram(bigram):
    key = tuple(bigram)
    if key not in bigram_counter:
        bigram_counter[key] = 1
    else:
        bigram_counter[key] += 1
```

A medida que recorremos el libro, tenemos que llevar la cuenta de cada par de palabras consecutivas. Así que si vemos la secuencia “man is not truly one”, añadiríamos los bigramas “man is”, “is not”, “not truly”, y así sucesivamente.

Para llevar la cuenta de estos bigramas, usaremos una lista llamada `window`, porque es como una ventana que se desliza sobre las páginas del libro, mostrando solo dos palabras a la vez. Inicialmente, `window` está vacía.

```
window = []
```

Usaremos la siguiente función para procesar las palabras una por una.

```
def process_word(word):
    window.append(word)

    if len(window) == 2:
        count_bigram(window)
        window.pop(0)
```

La primera vez que se llama a esta función, añade la palabra dada a `window`. Como solo hay una palabra en la window, todavía no tenemos un bigrama, así que la función termina.

La segunda vez que se llama – y todas las veces después – añade una segunda palabra a `window`. Como hay dos palabras en la window, llama a `count_bigram` para llevar la cuenta de cuántas veces aparece cada bigrama. Luego usa `pop` para eliminar la primera palabra de la window.

El siguiente programa hace un bucle por las palabras del libro y las procesa una a una.

```
for line in open(filename):
    for word in split_line(line):
        word = clean_word(word)
        process_word(word)
```

El resultado es un diccionario que asocia cada bigrama con el número de veces que aparece. Podemos usar `print_most_common` para ver los bigramas más comunes.

```
print_most_common(bigram_counter)
```

Al mirar estos resultados, podemos hacernos una idea de qué pares de palabras tienen más probabilidad de aparecer juntos. También podemos usar los resultados para generar texto aleatorio, así.

```
random.seed(0)
```

```
bigrams = list(bigram_counter)
weights = bigram_counter.values()
random_bigrams = random.choices(bigrams, weights=weights, k=6)
```

`bigrams` es una lista de los bigramas que aparecen en los libros. `weights` es una lista de sus frecuencias, así que `random_bigrams` es una muestra donde la probabilidad de que se seleccione un bigrama es proporcional a su frecuencia.

Aquí están los resultados.

```
for pair in random_bigrams:
    print(' '.join(pair), end=' ')
```

Esta forma de generar texto es mejor que elegir palabras aleatorias, pero todavía no tiene mucho sentido.

12.8. Análisis de Markov

Podemos hacerlo mejor con el análisis de texto mediante cadenas de Markov, que calcula, para cada palabra de un texto, la lista de palabras que vienen después. Como ejemplo, analizaremos esta letra de la canción de Monty Python *Eric, the Half a Bee*:

```
song = """
Half a bee, philosophically,
Must, ipso facto, half not be.
But half the bee has got to be
Vis a vis, its entity. D'you see?
"""
```

Para guardar los resultados, usaremos un diccionario que asocia cada palabra con la lista de palabras que la siguen.

```
successor_map = {}
```

Como ejemplo, empecemos con las dos primeras palabras de la canción.

```
first = 'half'
second = 'a'
```

Si la primera palabra no está en `successor_map`, tenemos que añadir un nuevo elemento que asocie la primera palabra con una lista que contiene la segunda palabra.

```
successor_map[first] = [second]
successor_map
```

Si la primera palabra ya está en el diccionario, podemos buscarla para obtener la lista de sucesores que hemos visto hasta ahora, y añadir el nuevo.

```
first = 'half'
second = 'not'

successor_map[first].append(second)
successor_map
```

La siguiente función encapsula estos pasos.

```
def add_bigram(bigram):
    first, second = bigram

    if first not in successor_map:
        successor_map[first] = [second]
    else:
        successor_map[first].append(second)
```

Si el mismo bigrama aparece más de una vez, la segunda palabra se añade a la lista más de una vez. De esta manera, `successor_map` lleva la cuenta de cuántas veces aparece cada sucesor.

Como hicimos en la sección anterior, usaremos una lista llamada `window` para guardar pares de palabras consecutivas. Y usaremos la siguiente función para procesar las palabras una por una.

```
def process_word_bigram(word):
    window.append(word)

    if len(window) == 2:
        add_bigram(window)
        window.pop(0)
```

Así es como la usamos para procesar las palabras de la canción.

```
successor_map = {}
window = []

for word in song.split():
    word = clean_word(word)
    process_word_bigram(word)
```

Y aquí están los resultados.

```
successor_map
```

La palabra `'half'` puede ir seguida de `'a'`, `'not'` o `'the'`. La palabra `'a'` puede ir seguida de `'bee'` o `'vis'`. La mayoría de las demás palabras aparecen solo una vez, así que van seguidas de una única palabra.

Ahora analicemos el libro.

```
successor_map = {}
window = []

for line in open(filename):
    for word in split_line(line):
        word = clean_word(word)
        process_word_bigram(word)
```

Podemos buscar cualquier palabra y encontrar las palabras que pueden seguirla.

```
# I used this cell to find a predecessor with a good number of possible successors
# and at least one repeated word.

def has_duplicates(t):
    return len(set(t)) < len(t)

for key, value in successor_map.items():
    if len(value) == 7 and has_duplicates(value):
        print(key, value)
```

```
successor_map['going']
```

En esta lista de sucesores, observa que la palabra `'to'` aparece tres veces – los demás sucesores aparecen solo una vez.

12.9. Generar texto

Podemos usar los resultados de la sección anterior para generar texto nuevo con las mismas relaciones entre palabras consecutivas que en el original. Así funciona:

- Empezando con cualquier palabra que aparezca en el texto, buscamos sus posibles sucesores y elegimos uno al azar.
- Luego, usando la palabra elegida, buscamos sus posibles sucesores y elegimos uno al azar.

Podemos repetir este proceso para generar tantas palabras como queramos. Como ejemplo, empecemos con la palabra `'although'`. Estas son las palabras que pueden seguirla.

```
word = 'although'
successors = successor_map[word]
successors
```

```
# this cell initializes the random number generator so it
# starts at the same point in the sequence each time this
# notebook runs.

random.seed(2)
```

Podemos usar `choice` para elegir de la lista con la misma probabilidad.

```
word = random.choice(successors)
word
```

Si la misma palabra aparece más de una vez en la lista, es más probable que sea seleccionada.

Repitiendo estos pasos, podemos usar el siguiente bucle para generar una serie más larga.

```
for i in range(10):
    successors = successor_map[word]
    word = random.choice(successors)
    print(word, end=' ')
```

El resultado suena más como una oración real, pero todavía no tiene mucho sentido.

Podemos hacerlo mejor usando más de una palabra como clave en `successor_map`. Por ejemplo, podemos crear un diccionario que asocie cada bigrama – o trigram – con la lista de palabras que vienen después. Como ejercicio, tendrás la oportunidad de implementar este análisis y ver cómo son los resultados.

12.10. Depuración

A estas alturas estamos escribiendo programas más sustanciales, y puede que descubras que pasas más tiempo depurando. Si estás atascado con un bug difícil, aquí tienes algunas cosas que

puedes probar:

- Leer: examina tu código, léelo en voz alta para ti y comprueba que dice lo que querías decir.
- Ejecutar: experimenta haciendo cambios y ejecutando versiones diferentes. A menudo, si muestras lo correcto en el lugar correcto del programa, el problema se vuelve obvio, pero a veces tienes que construir código de apoyo.
- Rumiar: ¡tómate tiempo para pensar! ¿Qué tipo de error es: de sintaxis, de runtime, o semántico? ¿Qué información puedes obtener de los mensajes de error, o del output del programa? ¿Qué tipo de error podría causar el problema que estás viendo? ¿Qué cambiaste por última vez, antes de que apareciera el problema?
- Rubberducking: si explicas el problema a otra persona, a veces encuentras la respuesta antes de terminar de hacer la pregunta. A menudo no necesitas a la otra persona; podrías hablarle a un patito de goma. Y ese es el origen de la estrategia conocida como **rubber duck depuración**. No me lo estoy inventando – ver https://en.wikipedia.org/wiki/Rubber_duck_depuración.
- Retirarse: en algún momento, lo mejor es retroceder – deshacer cambios recientes – hasta llegar a un programa que funcione. Luego puedes empezar a reconstruir.
- Descansar: si le das un respiro a tu cerebro, a veces encontrará el problema por ti.

Los programadores principiantes a veces se quedan atascados en una de estas actividades y olvidan las demás. Cada actividad tiene su propio modo de fallo.

Por ejemplo, leer tu código funciona si el problema es un error tipográfico, pero no si el problema es un malentendido conceptual. Si no entiendes lo que hace tu programa, puedes leerlo 100 veces y nunca ver el error, porque el error está en tu cabeza.

Ejecutar experimentos puede funcionar, especialmente si ejecutas tests pequeños y simples. Pero si haces experimentos sin pensar ni leer tu código, puede llevar mucho tiempo averiguar qué está pasando.

Tienes que tomarte tiempo para pensar. Depurar es como una ciencia experimental. Deberías tener al menos una hipótesis sobre cuál es el problema. Si hay dos o más posibilidades, intenta pensar en un test que elimine una de ellas.

Pero incluso las mejores técnicas de depuración fallarán si hay demasiados errores, o si el código que intentas arreglar es demasiado grande y complicado. A veces la mejor opción es retirarse, simplificando el programa hasta volver a algo que funcione.

Los programadores principiantes a menudo se resisten a retirarse porque no soportan borrar una línea de código (aunque esté mal). Si te hace sentir mejor, copia tu programa en otro archivo antes de empezar a recortarlo. Luego puedes copiar las piezas de vuelta una por una.

Encontrar un bug difícil requiere leer, ejecutar, rumiar, retirarse y a veces descansar. Si te atascas en una de estas actividades, prueba las otras.

12.11. Glosario

valor por defecto: El valor asignado a un parámetro si no se proporciona ningún argumento.

override: Reemplazar un valor por defecto por un argumento.

determinista: Un programa determinista hace lo mismo cada vez que se ejecuta, dados los mismos inputs.

pseudorandom: Una secuencia pseudorandom de números parece aleatoria, pero es generada por un programa determinista.

bigrama: Una secuencia de dos elementos, a menudo palabras.

trigram: Una secuencia de tres elementos.

n-gram: Una secuencia de un número no especificado de elementos.

rubber duck depuración: Una forma de depurar explicando un problema en voz alta a un objeto inanimado.

12.12. Ejercicios

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

12.12.1. Pregunta a un asistente virtual

En `add_bigram`, la sentencia `if` crea una nueva lista o añade un elemento a una lista existente, dependiendo de si la clave ya está en el diccionario.

```
def add_bigram(bigram):
    first, second = bigram

    if first not in successor_map:
        successor_map[first] = [second]
    else:
        successor_map[first].append(second)
```

Diccionarios proporcionan un método llamado `setdefault` que podemos usar para hacer lo mismo de forma más concisa. Pregunta a un asistente virtual cómo funciona, o copia `add_bigram` en un asistente virtual y pregunta "Can you rewrite this using `setdefault`?"

En este capítulo implementamos análisis y generación de texto con cadenas de Markov. Si tienes curiosidad, puedes pedirle a un asistente virtual más información sobre el tema. Una de las cosas que podrías aprender es que los asistentes virtuales usan algoritmos que son similares en muchos aspectos – pero también diferentes en aspectos importantes. Pregunta a un VA: «What are the differences between large language models like GPT and Markov chain text analysis?

12.12.2. Ejercicio

Escribe una función que cuente el número de veces que aparece cada trigram (secuencia de tres palabras). Si pruebas tu función con el texto de *Dr. Jekyll and Mr. Hyde*, deberías encontrar que el trigram más común es "said the lawyer".

Pista: escribe una función llamada `count_trigram` que sea similar a `count_bigram`. Luego escribe una función llamada `process_word_trigram` que sea similar a `process_word_bigram`.

Puedes usar el siguiente bucle para leer el libro y procesar las palabras.

```
trigram_counter = {}
window = []

for line in open(filename):
    for word in split_line(line):
        word = clean_word(word)
        process_word_trigram(word)
```

Luego usa `print_most_common` para encontrar los trigrams más comunes del libro.

```
print_most_common(trigram_counter)
```

12.12.3. Ejercicio

Ahora implementemos análisis de texto con cadenas de Markov usando un mapeo de cada bigrama a una lista de posibles sucesores.

Empezando con `add_bigram`, escribe una función llamada `add_trigram` que tome una lista de tres palabras y añada o actualice un elemento en `successor_map`, usando las dos primeras palabras como clave y la tercera palabra como posible sucesor.

Aquí tienes una versión de `process_word_trigram` que llama a `add_trigram`.

```
def process_word_trigram(word):
    window.append(word)

    if len(window) == 3:
        add_trigram(window)
        window.pop(0)
```

Puedes usar el siguiente bucle para probar tu función con la letra de “Eric, the Half a Bee”.

```
successor_map = {}
window = []

for string in song.split():
    word = string.strip(punctuation).lower()
    process_word_trigram(word)
```

Si tu función funciona como se espera, el predecesor `('half', 'a')` debería asociarse con una lista cuyo único elemento es `'bee'`. De hecho, resulta que cada bigrama de esta canción aparece solo una vez, así que todos los valores de `successor_map` tienen un único elemento.

```
successor_map
```

Puedes usar el siguiente bucle para probar tu función con las palabras del libro.

```
successor_map = {}
window = []

for line in open(filename):
    for word in split_line(line):
        word = clean_word(word)
        process_word_trigram(word)
```

En el siguiente ejercicio, usarás los resultados para generar nuevo texto aleatorio.

12.12.4. Ejercicio

Para este ejercicio, asumiremos que `successor_map` es un diccionario que asocia cada bigrama con la lista de palabras que lo siguen.

```
# this cell initializes the random number generator so it
# starts at the same point in the sequence each time this
# notebook runs.

random.seed(3)
```

Para generar texto aleatorio, empezaremos eligiendo una clave aleatoria de `successor_map`.

```
successors = list(successor_map)
bigram = random.choice(successors)
bigram
```

Ahora escribe un bucle que genere 50 palabras más siguiendo estos pasos:

1. En `successor_map`, busca la lista de palabras que pueden seguir a `bigram`.
2. Elige una de ellas al azar e imprímela.

3. Para la siguiente iteración, crea un nuevo bigrama que contenga la segunda palabra de `bigram` y el sucesor elegido.

Por ejemplo, si empezamos con el bigrama `('doubted', 'if')` y elegimos `'from'` como sucesor, el siguiente bigrama es `('if', 'from')`.

Si todo funciona, deberías encontrar que el texto generado es reconociblemente similar en estilo al original, y algunas frases tienen sentido, pero el texto podría saltar de un tema a otro.

Como ejercicio extra, modifica tu solución a los dos últimos ejercicios para usar trigrams como claves en `successor_map`, y mira qué efecto tiene en los resultados.

Think Python: 3rd Edition

Traducción al español por midudev (Miguel Ángel Durán).

Puedes pedir las versiones impresa y ebook de *Think Python 3e* en Bookshop.org y Amazon.

Crédito: las fotos se descargaron de Lorem Picsum, un servicio que proporciona imágenes de marcador de posición. El nombre es una referencia a “lorem ipsum”, que es un nombre para texto de marcador de posición.

```
# This cell downloads an archive file that contains the the files we'll
# use for the examples in this chapter.

download('https://github.com/AllenDowney/ThinkPython/raw/v3/photos.zip');
```

```
# WARNING: This cell removes the photos/ directory if it already exists.
# Any files already in the photos/ directory will be deleted.

# !rm -rf photos/
```

```
!unzip -o photos.zip
```

13. Archivos y bases de datos

La mayoría de los programas que hemos visto hasta ahora son **efímeros** en el sentido de que se ejecutan durante poco tiempo y producen output, pero cuando terminan, sus datos desaparecen. Cada vez que ejecutas un programa efímero, empieza desde cero.

Otros programas son **persistentes**: se ejecutan durante mucho tiempo (o todo el tiempo); mantienen al menos parte de sus datos en almacenamiento a largo plazo; y, si se cierran y se reinician, continúan donde lo dejaron.

Una forma sencilla de que los programas mantengan sus datos es leer y escribir archivos de texto. Una alternativa más versátil es almacenar datos en una base de datos. Las bases de datos son archivos especializados que se pueden leer y escribir de forma más eficiente que los archivos de texto, y proporcionan capacidades adicionales.

En este capítulo escribiremos programas que leen y escriben archivos de texto y bases de datos, y como ejercicio escribirás un programa que busca duplicados en una colección de fotos. Pero antes de poder trabajar con un archivo, tienes que encontrarlo, así que empezaremos con nombres de archivo, rutas y directorios.

13.1. Nombres de archivo y rutas

Los archivos se organizan en **directorios**, también llamados "carpetas". Todo programa en ejecución tiene un **directorio de trabajo actual**, que es el directorio por defecto para la mayoría de operaciones. Por ejemplo, cuando abres un archivo, Python lo busca en el directorio de trabajo actual.

El módulo `os` proporciona funciones para trabajar con archivos y directorios ("os" significa "operating system"). Proporciona una función llamada `getcwd` que obtiene el nombre del directorio de trabajo actual.

```
# this cell replaces `os.cwd` with a function that returns a fake path

import os

def getcwd():
    return "/home/dinsdale"

os.getcwd = getcwd
```

```
import os

os.getcwd()
```

El resultado en este ejemplo es el directorio personal de un usuario llamado `dinsdale`. Una string como `'/home/dinsdale'` que identifica un archivo o directorio se llama **ruta**.

Un nombre de archivo sencillo como `'memo.txt'` también se considera una ruta, pero es un **ruta relativa** porque especifica un nombre de archivo relativo al directorio actual. En este ejemplo, el directorio actual es `/home/dinsdale`, así que `'memo.txt'` equivale a la ruta completa `'/home/dinsdale/memo.txt'`.

Un ruta que empieza con `/` no depende del directorio actual – se llama **ruta absoluta**. Para encontrar la ruta absoluta de un archivo, puedes usar `abspath`.

```
os.path.abspath('memo.txt')
```

El módulo `os` proporciona otras funciones para trabajar con nombres de archivo y rutas. `listdir` devuelve una lista con el contenido del directorio indicado, incluyendo archivos y otros directorios. Aquí tienes un ejemplo que lista el contenido de un directorio llamado `photos`.

```
os.listdir('photos')
```

Este directorio contiene un archivo de texto llamado `notes.txt` y tres directorios. Los directorios contienen archivos de imagen en formato JPEG.

```
os.listdir('photos/jan-2023')
```

Para comprobar si existe un archivo o directorio, podemos usar `os.path.exists`.

```
os.path.exists('photos')
```

```
os.path.exists('photos/apr-2023')
```

Para comprobar si una ruta se refiere a un archivo o a un directorio, podemos usar `isdir`, que devuelve `True` si una ruta se refiere a un directorio.

```
os.path.isdir('photos')
```

Y `isfile`, que devuelve `True` si una ruta se refiere a un archivo.

```
os.path.isfile('photos/notes.txt')
```

Un desafío al trabajar con rutas es que se ven distintos en diferentes sistemas operativos. En macOS y sistemas UNIX como Linux, los nombres de directorios y archivos en una ruta se separan con una barra inclinada, `/`. Windows usa una barra invertida, `\`. Así que, si ejecutas estos ejemplos en Windows, verás barras invertidas en los rutas y tendrás que reemplazar las barras inclinadas de los ejemplos.

O, para escribir código que funcione en ambos sistemas, puedes usar `os.path.join`, que une nombres de directorio y de archivo en una ruta usando una barra inclinada o invertida, según el sistema operativo que estés usando.

```
os.path.join('photos', 'jan-2023', 'photo1.jpg')
```

Más adelante en este capítulo usaremos estas funciones para buscar en un conjunto de directorios y encontrar todos los archivos de imagen.

13.2. f-strings

Una forma en que los programas almacenan datos es escribirlos en un archivo de texto. Por ejemplo, supón que observas camellos y quieres registrar el número de camellos que has visto durante un periodo de observación. Y supón que en un año y medio has visto `23` camellos. Los datos de tu cuaderno de observación de camellos podrían verse así.

```
num_years = 1.5
num_camels = 23
```

Para escribir estos datos en un archivo, puedes usar el método `write`, que vimos en el Capítulo 8. El argumento de `write` tiene que ser una string, así que si queremos poner otros valores en un archivo, tenemos que convertirlos a strings. La forma más sencilla de hacerlo es con la función integrada `str`.

Así es como se ve:

```
writer = open('camel-spotting-book.txt', 'w')
writer.write(str(num_years))
writer.write(str(num_camels))
writer.close()
```

Eso funciona, pero `write` no añade un espacio ni una nueva línea a menos que lo incluyas explícitamente. Si volvemos a leer el archivo, vemos que los dos números quedan pegados.

```
open('camel-spotting-book.txt').read()
```

Como mínimo, deberíamos añadir espacios en blanco entre los números. Y ya que estamos, añadamos algo de texto explicativo.

Para escribir una combinación de strings y otros valores, podemos usar una **f-string**, que es una string que tiene la letra `f` antes de la comilla de apertura y contiene una o más expresiones de Python entre llaves. La siguiente f-string contiene una expresión, que es el nombre de una variable.

```
f'I have spotted {num_camels} camels'
```

El resultado es una string donde la expresión se ha evaluado y se ha reemplazado por el resultado. Puede haber más de una expresión.

```
f'In {num_years} years I have spotted {num_camels} camels'
```

Y las expresiones pueden contener operadores y llamadas a función.

```
line = f'In {round(num_years * 12)} months I have spotted {num_camels} camels'
line
```

Así que podríamos escribir los datos en un archivo de texto así.

```
writer = open('camel-spotting-book.txt', 'w')
writer.write(f'Years of observation: {num_years}\n')
writer.write(f'Camels spotted: {num_camels}\n')
writer.close()
```

Ambas f-strings terminan con la secuencia `\n`, que añade un carácter de nueva línea.

Podemos volver a leer el archivo así:

```
data = open('camel-spotting-book.txt').read()
print(data)
```

En una f-string, una expresión entre llaves se convierte en una string, así que puedes incluir listas, diccionarios y otros tipos.

```
t = [1, 2, 3]
d = {'one': 1}
f'Here is a list {t} and a dictionary {d}'
```

Si una f-string contiene una expresión no válida, el resultado es un error.

```
f'This is not a valid expression {t + 2}'
```

13.3. YAML

Una de las razones por las que los programas leen y escriben archivos es para almacenar **datos de configuración**, que son información que especifica qué debe hacer el programa y cómo.

Por ejemplo, en un programa que busca fotos duplicadas, podríamos tener un diccionario llamado `config` que contiene el nombre del directorio donde buscar, el nombre de otro directorio donde

debería almacenar los resultados y una lista de extensiones de archivo que debería usar para identificar archivos de imagen.

Así podría verse:

```
config = {
    'photo_dir': 'photos',
    'data_dir': 'photo_info',
    'extensions': ['jpg', 'jpeg'],
}
```

Para escribir estos datos en un archivo de texto, podríamos usar f-strings, como en la sección anterior. Pero es más fácil usar un módulo llamado `yaml` que está diseñado justo para este tipo de cosas.

El módulo `yaml` proporciona funciones para trabajar con archivos YAML, que son archivos de texto con un formato pensado para que sean fáciles de leer y escribir tanto para humanos *como* para programas.

Aquí tienes un ejemplo que usa la función `dump` para escribir el diccionario `config` en un archivo YAML.

```
# this cell installs the pyyaml package, which provides the yaml module

try:
    import yaml
except ImportError:
    !pip install pyyaml
```

```
import yaml

config_filename = 'config.yaml'
writer = open(config_filename, 'w')
yaml.dump(config, writer)
writer.close()
```

Si volvemos a leer el contenido del archivo, podemos ver cómo es el formato YAML.

```
readback = open(config_filename).read()
print(readback)
```

Ahora podemos usar `safe_load` para volver a leer el archivo YAML.

```
reader = open(config_filename)
config_readback = yaml.safe_load(reader)
config_readback
```

El resultado es un nuevo diccionario que contiene la misma información que el original, pero no es el mismo diccionario.

```
config is config_readback
```

Convertir un objeto como un diccionario en una string se llama **serialización**. Convertir la string de vuelta en un objeto se llama **deserialización**. Si serializas y luego deserializas un objeto, el resultado debería ser equivalente al original.

13.4. Shelve

Hasta ahora hemos estado leyendo y escribiendo archivos de texto – ahora consideremos las bases de datos. Una **base de datos** es un archivo organizado para almacenar datos. Algunas bases de datos se organizan como una tabla con filas y columnas de información. Otras se organizan como un diccionario que mapea claves a valores; a veces se llaman **almacenes clave-valor**.

El módulo `shelve` proporciona funciones para crear y actualizar un almacén clave-valor llamado "shelf". Como ejemplo, crearemos un shelf para contener captions de las figuras del directorio `photos`. Usaremos el diccionario `config` para obtener el nombre del directorio donde deberíamos poner el shelf.

```
config['data_dir']
```

Podemos usar `os.makedirs` para crear este directorio, si todavía no existe.

```
os.makedirs(config['data_dir'], exist_ok=True)
```

Y `os.path.join` para crear una ruta que incluya el nombre del directorio y el nombre del archivo del shelf, `captions`.

```
db_file = os.path.join(config['data_dir'], 'captions')
db_file
```

Ahora podemos usar `shelve.open` para abrir el archivo del shelf. El argumento `c` indica que el archivo debería crearse si es necesario.

```
import shelve

db = shelve.open(db_file, 'c')
db
```

Si obtienes un error como `db type could not be determined`, la causa más probable es que ya exista un archivo con el mismo nombre, pero que no sea una base de datos shelve válida (por ejemplo, puede estar corrupto o haber sido creado por otra cosa).

En ese caso, la solución más sencilla es eliminar el archivo existente y volver a ejecutar el código para que `shelve.open` pueda crear una base de datos nueva.

El valor de retorno es oficialmente un objeto `DbfilenameShelf`, llamado de manera más informal un objeto shelf.

El objeto shelf se comporta de muchas maneras como un diccionario. Por ejemplo, podemos usar el operador de corchetes para añadir un elemento, que es un mapeo de una clave a un valor.

```
key = 'jan-2023/photo1.jpg'
db[key] = 'Cat nose'
```

En este ejemplo, la clave es la ruta a un archivo de imagen y el valor es una string que describe la imagen.

También usamos el operador de corchetes para buscar una clave y obtener el valor correspondiente.

```
value = db[key]
value
```

Si haces otra asignación a una clave existente, `shelve` reemplaza el valor anterior.

```
db[key] = 'Close up view of a cat nose'
db[key]
```

Algunos métodos de diccionario, como `keys`, `values` e `items`, también funcionan con objetos shelf.

```
list(db.keys())
```

```
list(db.values())
```

Podemos usar el operador `in` para comprobar si una clave aparece en el shelf.

```
key in db
```

Y podemos usar una sentencia `for` para iterar sobre las claves.

```
for key in db:
    print(key, ':', db[key])
```

Como con otros archivos, deberías cerrar la base de datos cuando termines.

```
db.close()
```

Ahora, si listamos el contenido del directorio de datos, vemos dos archivos.

```
# When you open a shelve file, a backup file is created that has the suffix `.bak`.
# If you run this notebook more than once, you might see that file left behind.
# This cell removes it so the output shown in the book is correct.

!rm -f photo_info/captions.bak
```

```
os.listdir(config['data_dir'])
```

`captions.dat` contiene los datos que acabamos de almacenar. `captions.dir` contiene información sobre la organización de la base de datos que hace que el acceso sea más eficiente. El sufijo `dir` significa "directorio", pero no tiene nada que ver con los directorios con los que hemos estado trabajando y que contienen archivos.

13.5. Almacenar estructuras de datos

En el ejemplo anterior, las claves y valores del shelf son strings. Pero también podemos usar un shelf para contener estructuras de datos como listas y diccionarios.

Como ejemplo, volvamos al ejemplo de anagramas de un ejercicio en el Capítulo 11. Recuerda que hicimos un diccionario que mapea una string ordenada de letras a la lista de palabras que se pueden formar con esas letras. Por ejemplo, la clave `'opst'` mapea a la lista `['opts', 'post', 'pots', 'spot', 'stop', 'tops']`.

Usaremos la siguiente función para ordenar las letras de una palabra.

```
def sort_word(word):
    return ''.join(sorted(word))
```

Y aquí tienes un ejemplo.

```
word = 'pots'
key = sort_word(word)
key
```

Ahora abramos un shelf llamado `anagram_map`. El argumento `'n'` significa que siempre deberíamos crear un shelf nuevo y vacío, aunque ya exista uno.

```
db = shelve.open('anagram_map', 'n')
```

Ahora podemos añadir un elemento al shelf así.

```
db[key] = [word]
db[key]
```

En este elemento, la clave es una string y el valor es una lista de strings.

Ahora supón que encontramos otra palabra que contiene las mismas letras, como `tops`

```
word = 'tops'
key = sort_word(word)
key
```

La clave es la misma que en el ejemplo anterior, así que queremos añadir una segunda palabra a la misma lista de strings. Así es como lo haríamos si `db` fuera un diccionario.

```
db[key].append(word)          # INCORRECT
```

Pero si lo ejecutamos y luego buscamos la clave en el shelf, parece que no se ha actualizado.

```
db[key]
```

Este es el problema: cuando buscamos la clave, obtenemos una lista de strings, pero si modificamos la lista de strings, eso no afecta al shelf. Si queremos actualizar el shelf, tenemos que leer el valor antiguo, actualizarlo y luego escribir el nuevo valor de vuelta en el shelf.

```
anagram_list = db[key]
anagram_list.append(word)
db[key] = anagram_list
```

Ahora el valor del shelf está actualizado.

```
db[key]
```

Como ejercicio, puedes terminar este ejemplo leyendo la lista de palabras y almacenando todos los anagramas en un shelf.

```
db.close()
```

13.6. Comprobar archivos equivalentes

Ahora volvamos al objetivo de este capítulo: buscar archivos diferentes que contienen los mismos datos. Una forma de comprobarlo es leer el contenido de ambos archivos y compararlo.

Si los archivos contienen imágenes, tenemos que abrirlos con el modo `'rb'`, donde `'r'` significa que queremos leer el contenido y `'b'` indica **modo binario**. En modo binario, el contenido no se interpreta como texto – se trata como una secuencia de bytes.

Aquí tienes un ejemplo que abre y lee un archivo de imagen.

```
path1 = 'photos/jan-2023/photo1.jpg'
data1 = open(path1, 'rb').read()
type(data1)
```

El resultado de `read` es un objeto `bytes` – como sugiere el nombre, contiene una secuencia de bytes.

En general, el contenido de un archivo de imagen no es legible para humanos. Pero si leemos el contenido de un segundo archivo, podemos usar el operador `==` para comparar.

```
path2 = 'photos/jan-2023/photo2.jpg'
data2 = open(path2, 'rb').read()
data1 == data2
```

Estos dos archivos no son equivalentes.

Encapsulemos lo que tenemos hasta ahora en una función.

```
def same_contents(path1, path2):
    data1 = open(path1, 'rb').read()
    data2 = open(path2, 'rb').read()
    return data1 == data2
```

Si solo tenemos dos archivos, esta función es una buena opción. Pero supón que tenemos un gran número de archivos y queremos saber si dos cualesquiera contienen los mismos datos. Sería ineficiente comparar cada par de archivos.

Una alternativa es usar una **función hash**, que toma el contenido de un archivo y calcula un **digest**, que normalmente es un entero grande. Si dos archivos contienen los mismos datos, tendrán el mismo digest. Si dos archivos son diferentes, *casi siempre* tendrán digests diferentes.

El módulo `hashlib` proporciona varias hash funciones – la que usaremos se llama `md5`. Empezaremos usando `hashlib.md5` para crear un objeto `HASH`.

```
import hashlib

md5_hash = hashlib.md5()
type(md5_hash)
```

El objeto `HASH` proporciona un método `update` que toma el contenido del archivo como argumento.

```
md5_hash.update(data1)
```

Ahora podemos usar `hexdigest` para obtener el digest como una string de dígitos hexadecimales que representan un entero en base 16.

```
digest = md5_hash.hexdigest()
digest
```

La siguiente función encapsula estos pasos.

```
def md5_digest(filename):
    data = open(filename, 'rb').read()
    md5_hash = hashlib.md5()
    md5_hash.update(data)
    digest = md5_hash.hexdigest()
    return digest
```

Si hacemos hash del contenido de un archivo diferente, podemos confirmar que obtenemos un digest distinto.

```
filename2 = 'photos/feb-2023/photo2.jpg'
md5_digest(filename2)
```

Ahora tenemos casi todo lo que necesitamos para encontrar archivos equivalentes. El último paso es buscar en un directorio y encontrar todos los archivos de imagen.

13.7. Recorrer directorios

La siguiente función toma como argumento el directorio donde queremos buscar. Usa `listdir` para iterar sobre el contenido del directorio. Cuando encuentra un archivo, imprime su ruta completa. Cuando encuentra un directorio, se llama a sí misma recursivamente para buscar en el subdirectorio.

```
def walk(dirname):
    for name in os.listdir(dirname):
        path = os.path.join(dirname, name)

        if os.path.isfile(path):
            print(path)
        elif os.path.isdir(path):
            walk(path)
```

Podemos usarla así:

```
walk('photos')
```

El orden de los resultados depende de detalles del sistema operativo.

13.8. Depuración

Cuando lees y escribes archivos, puedes encontrarte con problemas relacionados con espacios en blanco. Estos errores pueden ser difíciles de depurar porque los caracteres de espacio en blanco normalmente son invisibles. Por ejemplo, aquí tienes una string que contiene espacios, un tab representado por la secuencia `\t` y una nueva línea representada por la secuencia `\n`. Cuando la imprimimos, no vemos los caracteres de espacio en blanco.

```
s = '1 2\t 3\n 4'
print(s)
```

La función integrada `repr` puede ayudar. Toma cualquier objeto como argumento y devuelve una representación en string del objeto. Para strings, representa los caracteres de espacio en blanco con secuencias de barra invertida.

```
print(repr(s))
```

Esto puede ser útil para depurar.

Otro problema que puedes encontrar es que distintos sistemas usan distintos caracteres para indicar el final de una línea. Algunos sistemas usan una nueva línea, representada como `\n`. Otros usan un carácter de retorno, representado como `\r`. Algunos usan ambos. Si mueves archivos entre sistemas diferentes, estas inconsistencias pueden causar problemas.

La capitalización de los nombres de archivo es otro problema que puedes encontrar si trabajas con distintos sistemas operativos. En macOS y UNIX, los nombres de archivo pueden contener letras minúsculas y mayúsculas, dígitos y la mayoría de símbolos. Pero muchas aplicaciones de Windows ignoran la diferencia entre letras minúsculas y mayúsculas, y varios símbolos que se permiten en macOS y UNIX no se permiten en Windows.

13.9. Glosario

efímero: Un programa efímero normalmente se ejecuta durante poco tiempo y, cuando termina, sus datos se pierden.

persistente: Un programa persistente se ejecuta indefinidamente y mantiene al menos parte de sus datos en almacenamiento permanente.

directorio: Una colección de archivos y otros directorios.

directorio de trabajo actual: El directorio por defecto usado por un programa a menos que se especifique otro directorio.

ruta: Una string que especifica una secuencia de directorios, que a menudo conduce a un archivo.

ruta relativa: Un ruta que empieza desde el directorio de trabajo actual, o desde algún otro directorio especificado.

ruta absoluta: Un ruta que no depende del directorio actual.

f-string: Una string que tiene la letra `f` antes de la comilla de apertura y contiene una o más expresiones entre llaves.

datos de configuración: Datos, a menudo almacenados en un archivo, que especifican qué debe hacer un programa y cómo.

serialización: Convertir un objeto en una string.

deserialización: Convertir una string en un objeto.

base de datos: Un archivo cuyo contenido está organizado para realizar ciertas operaciones de manera eficiente.

almacenes clave-valor: Una base de datos cuyo contenido está organizado como un diccionario con claves que corresponden a valores.

modo binario: Una forma de abrir un archivo para que el contenido se interprete como una secuencia de bytes en lugar de como una secuencia de caracteres.

función hash: Una función que toma un objeto y calcula un entero, que a veces se llama digest.

digest: El resultado de una función hash, especialmente cuando se usa para comprobar si dos objetos son iguales.

13.10. Ejercicios

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

13.10.1. Pregunta a un asistente virtual

Hay varios temas que surgieron en este capítulo que no expliqué en detalle. Aquí tienes algunas preguntas que puedes hacerle a un asistente virtual para obtener más información.

- "¿Cuáles son las diferencias entre programas efímeros y persistentes?"
- "¿Cuáles son algunos ejemplos de programas persistentes?"

- "¿Cuál es la diferencia entre una ruta relativa y una ruta absoluta?"
- "¿Por qué el módulo `yaml` tiene funciones llamadas `load` y `safe_load`?"
- "Cuando escribo un shelf de Python, ¿qué son los archivos con sufijos `dat` y `dir`?"
- "Además de los almacenes clave-valor, ¿qué otros tipos de bases de datos existen?"
- "Cuando leo un archivo, ¿cuál es la diferencia entre modo binario y modo texto?"
- "¿Cuáles son las diferencias entre un objeto bytes y una string?"
- "¿Qué es una función hash?"
- "¿Qué es un digest MD5?"

Como siempre, si te quedas atascado en alguno de los ejercicios siguientes, considera pedir ayuda a un AV. Junto con tu pregunta, quizá quieras pegar las funciones relevantes de este capítulo.

13.10.2. Ejercicio

Escribe una función llamada `replace_all` que tome como argumentos una string patrón, una string de reemplazo y dos nombres de archivo. Debería leer el primer archivo y escribir el contenido en el segundo archivo (creándolo si es necesario). Si la string patrón aparece en cualquier parte del contenido, debería reemplazarse por la string de reemplazo.

Aquí tienes un esquema de la función para empezar.

```
def replace_all(old, new, source_path, dest_path):
    # read the contents of the source file
    reader = open(source_path)

    # replace the old string with the new

    # write the result into the destination file
```

Para probar tu función, lee el archivo `photos/notes.txt`, reemplaza `'photos'` por `'images'` y escribe el resultado en el archivo `photos/new_notes.txt`.

```
source_path = 'photos/notes.txt'
open(source_path).read()
```

```
dest_path = 'photos/new_notes.txt'
old = 'photos'
new = 'images'
replace_all(old, new, source_path, dest_path)
```

```
open(dest_path).read()
```

13.10.3. Ejercicio

En una sección anterior, usamos el módulo `shelve` para crear un almacén clave-valor que mapea una string ordenada de letras a una lista de anagramas. Para terminar el ejemplo, escribe una función llamada `add_word` que tome como argumentos una string y un objeto shelf.

Debería ordenar las letras de la palabra para crear una clave, y luego comprobar si la clave ya está en el shelf. Si no, debería crear una lista que contenga la nueva palabra y añadirla al shelf. Si sí, debería añadir la nueva palabra al valor existente.

Puedes usar este bucle para probar tu función.

```
download('https://raw.githubusercontent.com/AllenDowney/ThinkPython/v3/words.txt');
```

```
word_list = open('words.txt').read().split()

db = shelve.open('anagram_map', 'n')
for word in word_list:
    add_word(word, db)
```

Si todo funciona, deberías poder buscar una clave como `'opst'` y obtener una lista de palabras que se pueden formar con esas letras.

```
db['opst']
```

```
for key, value in db.items():
    if len(value) > 8:
        print(value)
```

```
db.close()
```

13.10.4. Ejercicio

En una colección grande de archivos, puede haber más de una copia del mismo archivo, almacenada en distintos directorios o con distintos nombres de archivo. El objetivo de este ejercicio es buscar duplicados. Como ejemplo, trabajaremos con archivos de imagen en el directorio `photos`.

Así es como funcionará:

- Usaremos la función `walk` de **`section_walking_directories`** para buscar en este directorio archivos que terminen con una de las extensiones en `config['extensions']`.
- Para cada archivo, usaremos `md5_digest` de **`section_md5_digest`** para calcular un digest del contenido.
- Usando un shelf, haremos un mapeo de cada digest a una lista de rutas con ese digest.
- Finalmente, buscaremos en el shelf cualquier digest que mapee a múltiples archivos.
- Si encontramos alguno, usaremos `same_contents` para confirmar que los archivos contienen los mismos datos.

Voy a sugerir algunas funciones para escribir primero, y luego lo juntaremos todo.

1. Para identificar archivos de imagen, escribe una función llamada `is_image` que tome una ruta y una lista de extensiones de archivo, y devuelva `True` si la ruta termina con una de las extensiones de la lista. Pista: usa `os.path.splitext` – o pide a un asistente virtual que escriba esta función por ti.

Puedes usar `doctest` para probar tu función.

```
from doctest import run_docstring_examples

def run_doctests(func):
    run_docstring_examples(func, globals(), name=func.__name__)

run_doctests(is_image)
```

2. Escribe una función llamada `add_path` que tome como argumentos una ruta y un shelf. Debería usar `md5_digest` para calcular un digest del contenido del archivo. Luego debería actualizar el shelf, ya sea creando un nuevo elemento que mapee el digest a una lista que contenga la ruta, o añadiendo la ruta a la lista si ya existe.

3. Escribe una versión de `walk` llamada `walk_images` que tome un directorio y recorra los archivos del directorio y sus subdirectorios. Para cada archivo, debería usar `is_image` para comprobar si es un archivo de imagen y `add_path` para añadirlo al shelf.

Cuando todo funcione, puedes usar el siguiente programa para crear el shelf, buscar en el directorio `photos` y añadir rutas al shelf, y luego comprobar si hay múltiples archivos con el mismo digest.

```
db = shelve.open('photos/digests', 'n')
walk_images('photos')

for digest, paths in db.items():
    if len(paths) > 1:
        print(paths)
```

Deberías encontrar un par de archivos que tienen el mismo digest. Usa `same_contents` para comprobar si contienen los mismos datos.

14. Clases y funciones

A estas alturas sabes cómo usar funciones para organizar código y cómo usar tipos integrados para organizar datos. El siguiente paso es la **programación orientada a objetos**, que usa tipos

definidos por el programador para organizar tanto código como datos.

La programación orientada a objetos es un tema amplio, así que avanzaremos gradualmente. En este capítulo, empezaremos con código que no es idiomático – es decir, no es el tipo de código que escriben los programadores con experiencia – pero es un buen punto de partida. En los dos capítulos siguientes, usaremos características adicionales para escribir código más idiomático.

14.1. Tipos definidos por el programador

Hemos usado muchos de los tipos integrados de Python – ahora definiremos un tipo nuevo. Como primer ejemplo, crearemos un tipo llamado `Time` que representa una hora del día. Un tipo definido por el programador también se llama **clase**. Una definición de clase se ve así:

```
class Time:
    """Represents a time of day."""
```

La cabecera indica que la nueva clase se llama `Time`. El cuerpo es un docstring que explica para qué sirve la clase. Definir una clase crea un **objeto de clase**.

El objeto de clase es como una fábrica para crear objetos. Para crear un objeto `Time`, llamas a `Time` como si fuera una función.

```
lunch = Time()
```

El resultado es un objeto nuevo cuyo tipo es `__main__.Time`, donde `__main__` es el nombre del módulo donde se define `Time`.

```
type(lunch)
```

Cuando imprimes un objeto, Python te dice de qué tipo es y dónde está almacenado en memoria (el prefijo `0x` significa que el número siguiente está en hexadecimal).

```
print(lunch)
```

Crear un objeto nuevo se llama **instanciación**, y el objeto es una **instancia** de la clase.

14.2. Atributos

Un objeto puede contener variables, que se llaman **atributos**. Podemos crear atributos usando notación de punto.

```
lunch.hour = 11
lunch.minute = 59
lunch.second = 1
```

Este ejemplo crea atributos llamados `hour`, `minute` y `second`, que contienen las horas, minutos y segundos de la hora `11:59:01`, que es la hora de comer por lo que a mí respecta.

El siguiente diagrama muestra el estado de `lunch` y sus atributos después de estas asignaciones.

```
from diagram import make_frame, make_binding

d1 = dict(hour=11, minute=59, second=1)
frame = make_frame(d1, name='Time', dy=-0.3, offsetx=0.48)
binding = make_binding('lunch', frame)
```

```
from diagram import diagram, adjust

width, height, x, y = [1.77, 1.24, 0.25, 0.86]
ax = diagram(width, height)
bbox = binding.draw(ax, x, y)
#adjust(x, y, bbox)
```

La variable `lunch` se refiere a un objeto `Time`, que contiene tres atributos. Cada atributo se refiere a un entero. Un diagrama de estado como este – que muestra un objeto y sus atributos – se llama **diagrama de objeto**.

Puedes leer el valor de un atributo usando el operador de punto.

```
lunch.hour
```

Puedes usar un atributo como parte de cualquier expresión.

```
total_minutes = lunch.hour * 60 + lunch.minute
total_minutes
```

Y puedes usar el operador de punto en una expresión dentro de una f-string.

```
f'{lunch.hour}:{lunch.minute}:{lunch.second}'
```

Pero observa que el ejemplo anterior no está en el formato estándar. Para arreglarlo, tenemos que imprimir los atributos `minute` y `second` con un cero inicial. Podemos hacerlo extendiendo las expresiones entre llaves con un **especificador de formato**. En el siguiente ejemplo, los especificadores de formato indican que `minute` y `second` deberían mostrarse con al menos dos dígitos y un cero inicial si hace falta.

```
f'{lunch.hour}:{lunch.minute:02d}:{lunch.second:02d}'
```

Usaremos esta f-string para escribir una función que muestra el valor de un objeto `Time`. Puedes pasar un objeto como argumento de la forma habitual. Por ejemplo, la siguiente función toma un objeto `Time` como argumento.

```
def print_time(time):
    s = f'{time.hour:02d}:{time.minute:02d}:{time.second:02d}'
    print(s)
```

Cuando la llamamos, podemos pasar `lunch` como argumento.

```
print_time(lunch)
```

14.3. Objetos como valores de retorno

Las funciones pueden devolver objetos. Por ejemplo, `make_time` toma parámetros llamados `hour`, `minute` y `second`, los almacena como atributos en un objeto `Time` y devuelve el objeto nuevo.

```
def make_time(hour, minute, second):
    time = Time()
    time.hour = hour
    time.minute = minute
    time.second = second
    return time
```

Puede sorprender que los parámetros tengan los mismos nombres que los atributos, pero esa es una forma habitual de escribir una función como esta. Así usamos `make_time` para crear un objeto `Time`.

```
time = make_time(11, 59, 1)
print_time(time)
```

14.4. Los objetos son mutables

Supón que vas a una proyección de una película, como *Monty Python and the Holy Grail*, que empieza a las `9:20 PM` y dura `92` minutos, que es una hora y `32` minutos. ¿A qué hora terminará la película?

Primero, crearemos un objeto `Time` que representa la hora de inicio.

```
start = make_time(9, 20, 0)
print_time(start)
```

Para encontrar la hora de fin, podemos modificar los atributos del objeto `Time`, añadiendo la duración de la película.

```
start.hour += 1
start.minute += 32
print_time(start)
```

La película terminará a las 10:52 PM.

Encapsulemos este cálculo en una función y generalicémoslo para que tome la duración de la película en tres parámetros: `hours`, `minutes` y `seconds`.

```
def increment_time(time, hours, minutes, seconds):
    time.hour += hours
    time.minute += minutes
    time.second += seconds
```

Aquí tienes un ejemplo que demuestra el efecto.

```
start = make_time(9, 20, 0)
increment_time(start, 1, 32, 0)
print_time(start)
```

El siguiente stack diagram muestra el estado del programa justo antes de que `increment_time` modifique el objeto.

```
from diagram import Frame, Binding, Value, Stack

d1 = dict(hour=9, minute=20, second=0)
obj1 = make_frame(d1, name='Time', dy=-0.25, offsetx=0.78)

binding1 = make_binding('start', frame, draw_value=False, dx=0.7)
frame1 = Frame([binding1], name='__main__', loc='left', offsetx=-0.2)

binding2 = Binding(Value('time'), draw_value=False, dx=0.7, dy=0.35)
binding3 = make_binding('hours', 1)
binding4 = make_binding('minutes',32)
binding5 = make_binding('seconds', 0)
frame2 = Frame([binding2, binding3, binding4, binding5], name='increment_time',
               loc='left', dy=-0.25, offsetx=0.08)

stack = Stack([frame1, frame2], dx=-0.3, dy=-0.5)
```

```
from diagram import Bbox

width, height, x, y = [3.4, 1.89, 1.75, 1.5]
ax = diagram(width, height)
bbox1 = stack.draw(ax, x, y)
bbox2 = obj1.draw(ax, x+0.23, y)
bbox = Bbox.union([bbox1, bbox2])
# adjust(x, y, bbox)
```

Dentro de la función, `time` es un alias de `start`, así que cuando se modifica `time`, cambia `start`.

Esta función funciona, pero después de ejecutarse nos quedamos con una variable llamada `start` que se refiere a un objeto que representa la hora de *fin*, y ya no tenemos un objeto que represente la hora de inicio. Sería mejor dejar `start` sin cambios y crear un objeto nuevo para representar la hora de fin. Podemos hacerlo copiando `start` y modificando la copia.

14.5. Copiar

El módulo `copy` proporciona una función llamada `copy` que puede duplicar cualquier objeto. Podemos importarla así.

```
from copy import copy
```

Para ver cómo funciona, empecemos con un objeto `Time` nuevo que representa la hora de inicio de la película.

```
start = make_time(9, 20, 0)
```

Y hagamos una copia.

```
end = copy(start)
```

Ahora `start` y `end` contienen los mismos datos.

```
print_time(start)
print_time(end)
```

Pero el operador `is` confirma que no son el mismo objeto.

```
start is end
```

Veamos qué hace el operador `==`.

```
start == end
```

Podrías esperar que `==` produzca `True` porque los objetos contienen los mismos datos. Pero para clases definidas por el programador, el comportamiento por defecto del operador `==` es el mismo que el del operador `is` – comprueba identidad, no equivalencia.

14.6. Funciones puras

Podemos usar `copy` para escribir funciones puras que no modifican sus parámetros. Por ejemplo, aquí tienes una función que toma un objeto `Time` y una duración en horas, minutos y segundos. Hace una copia del objeto original, usa `increment_time` para modificar la copia y la devuelve.

```
def add_time(time, hours, minutes, seconds):
    total = copy(time)
    increment_time(total, hours, minutes, seconds)
    return total
```

Así la usamos.

```
end = add_time(start, 1, 32, 0)
print_time(end)
```

El valor de retorno es un objeto nuevo que representa la hora de fin de la película. Y podemos confirmar que `start` no ha cambiado.

```
print_time(start)
```

`add_time` es una **función pura** porque no modifica ninguno de los objetos que se le pasan como argumentos y su único efecto es devolver un valor.

Cualquier cosa que se pueda hacer con funciones impuras también se puede hacer con funciones puras. De hecho, algunos lenguajes de programación solo permiten funciones puras. Los programas que usan funciones puras pueden ser menos propensos a errores, pero las funciones impuras a veces son convenientes y pueden ser más eficientes.

En general, te sugiero que escribas funciones puras siempre que sea razonable y recurras a funciones impuras solo si hay una ventaja convincente. Este enfoque podría llamarse un **estilo de programación funcional**.

14.7. Prototype and patch

En el ejemplo anterior, `increment_time` y `add_time` parecen funcionar, pero si probamos otro ejemplo veremos que no son del todo correctas.

Supón que llegas al cine y descubres que la película empieza a las `9:40`, no a las `9:20`. Esto es lo que ocurre cuando calculamos la hora de fin actualizada.

```
start = make_time(9, 40, 0)
end = add_time(start, 1, 32, 0)
print_time(end)
```

El resultado no es una hora válida. El problema es que `increment_time` no trata los casos en los que el número de segundos o minutos suma más de `60`.

Aquí tienes una versión mejorada que comprueba si `second` supera o iguala `60` – si es así, incrementa `minute` – y luego comprueba si `minute` supera o iguala `60` – si es así, incrementa `hour`.

```
def increment_time(time, hours, minutes, seconds):
    time.hour += hours
    time.minute += minutes
    time.second += seconds

    if time.second >= 60:
        time.second -= 60
        time.minute += 1

    if time.minute >= 60:
        time.minute -= 60
        time.hour += 1
```

Arreglar `increment_time` también arregla `add_time`, que la usa. Así que ahora el ejemplo anterior funciona correctamente.

```
end = add_time(start, 1, 32, 0)
print_time(end)
```

Pero esta función todavía no es correcta, porque los argumentos podrían ser mayores que `60`. Por ejemplo, supón que recibimos la duración como `92` minutos, en lugar de `1` hora y `32` minutos. Podríamos llamar a `add_time` así.

```
end = add_time(start, 0, 92, 0)
print_time(end)
```

El resultado no es una hora válida. Así que probemos un enfoque distinto, usando la función `divmod`. Haremos una copia de `start` y la modificaremos incrementando el atributo `minute`.

```
end = copy(start)
end.minute = start.minute + 92
end.minute
```

Ahora `minute` es `132`, que son `2` horas y `12` minutos. Podemos usar `divmod` para dividir por `60` y devolver el número de horas completas y el número de minutos sobrantes.

```
carry, end.minute = divmod(end.minute, 60)
carry, end.minute
```

Ahora `minute` es correcto, y podemos añadir las horas a `hour`.

```
end.hour += carry
print_time(end)
```

El resultado es una hora válida. Podemos hacer lo mismo con `hour` y `second`, y encapsular todo el proceso en una función.

```
def increment_time(time, hours, minutes, seconds):
    time.hour += hours
    time.minute += minutes
    time.second += seconds

    carry, time.second = divmod(time.second, 60)
    carry, time.minute = divmod(time.minute + carry, 60)
    carry, time.hour = divmod(time.hour + carry, 24)
```

Con esta versión de `increment_time`, `add_time` funciona correctamente, incluso si los argumentos superan `60`.

```
end = add_time(start, 0, 90, 120)
print_time(end)
```

Esta sección demuestra un plan de desarrollo de programas que llamo **prototype and patch**. Empezamos con un prototipo simple que funcionaba correctamente para el primer ejemplo. Luego

lo probamos con ejemplos más difíciles – cuando encontramos un error, modificamos el programa para corregirlo, como poner un parche en un neumático pinchado.

Este enfoque puede ser efectivo, especialmente si todavía no tienes una comprensión profunda del problema. Pero las correcciones incrementales pueden generar código innecesariamente complicado – porque trata muchos casos especiales – y poco fiable – porque es difícil saber si has encontrado todos los errores.

14.8. Design-first development

Un plan alternativo es **design-first development**, que implica más planificación antes de prototipar. En un proceso design-first, a veces una idea de alto nivel sobre el problema hace que la programación sea mucho más sencilla.

En este caso, la idea es que podemos pensar en un objeto `Time` como un número de tres dígitos en base 60 – también conocida como sexagesimal. El atributo `second` es la "columna de las unidades", el atributo `minute` es la "columna de los sesenta", y el atributo `hour` es la "columna de los treinta y seis cientos". Cuando escribimos `increment_time`, en realidad estábamos haciendo suma en base 60, por eso teníamos que llevar de una columna a la siguiente.

Esta observación sugiere otro enfoque para todo el problema – podemos convertir objetos `Time` en enteros y aprovechar que Python sabe hacer aritmética de enteros.

Aquí tienes una función que convierte de un `Time` a un entero.

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

El resultado es el número de segundos desde el comienzo del día. Por ejemplo, `01:01:01` es `1` hora, `1` minuto y `1` segundo desde el comienzo del día, que es la suma de `3600` segundos, `60` segundos y `1` segundo.

```
time = make_time(1, 1, 1)
print_time(time)
time_to_int(time)
```

Y aquí tienes una función que va en la otra dirección – convierte un entero en un objeto `Time` – usando la función `divmod`.

```
def int_to_time(seconds):
    minute, second = divmod(seconds, 60)
    hour, minute = divmod(minute, 60)
    return make_time(hour, minute, second)
```

Podemos probarla convirtiendo el ejemplo anterior de vuelta a un `Time`.

```
time = int_to_time(3661)
print_time(time)
```

Usando estas funciones, podemos escribir una versión más concisa de `add_time`.

```
def add_time(time, hours, minutes, seconds):
    duration = make_time(hours, minutes, seconds)
    seconds = time_to_int(time) + time_to_int(duration)
    return int_to_time(seconds)
```

La primera línea convierte los argumentos en un objeto `Time` llamado `duration`. La segunda línea convierte `time` y `duration` a segundos y los suma. La tercera línea convierte la suma en un objeto `Time` y lo devuelve.

Así funciona.

```
start = make_time(9, 40, 0)
end = add_time(start, 1, 32, 0)
print_time(end)
```

En cierto sentido, convertir de base 60 a base 10 y de vuelta es más difícil que simplemente tratar con horas. La conversión de base es más abstracta; nuestra intuición para trabajar con valores de tiempo es mejor.

Pero si tenemos la idea de tratar las horas como números en base 60 – e invertimos el esfuerzo de escribir las funciones de conversión `time_to_int` e `int_to_time` – obtenemos un programa más corto, más fácil de leer y depurar, y más fiable.

También es más fácil añadir características más adelante. Por ejemplo, imagina restar dos objetos `Time` para encontrar la duración entre ellos. El enfoque ingenuo es implementar la resta con préstamo. Usar las funciones de conversión es más fácil y tiene más probabilidades de ser correcto.

Irónicamente, a veces hacer un problema más difícil – o más general – lo hace más fácil, porque hay menos casos especiales y menos oportunidades de error.

14.9. Depuración

Python proporciona varias funciones integradas que son útiles para probar y depurar programas que trabajan con objetos. Por ejemplo, si no estás seguro de qué tipo es un objeto, puedes preguntarlo.

```
type(start)
```

También puedes usar `isinstance` para comprobar si un objeto es una instancia de una clase concreta.

```
isinstance(end, Time)
```

Si no estás seguro de si un objeto tiene un atributo concreto, puedes usar la función integrada `hasattr`.

```
hasattr(start, 'hour')
```

Para obtener todos los atributos, y sus valores, en un diccionario, puedes usar `vars`.

```
vars(start)
```

El módulo `structshape`, que vimos en el Capítulo 11, también funciona con tipos definidos por el programador.

```
download('https://raw.githubusercontent.com/AllenDowney/ThinkPython/v3/structshape.
```

```
from structshape import structshape

t = start, end
structshape(t)
```

14.10. Glosario

programación orientada a objetos: Un estilo de programación que usa objetos para organizar código y datos.

clase: Un tipo definido por el programador. Una definición de clase crea un nuevo objeto de clase.

objeto de clase: Un objeto que representa una clase – es el resultado de una definición de clase.

instanciación: El proceso de crear un objeto que pertenece a una clase.

instancia: Un objeto que pertenece a una clase.

atributo: Una variable asociada con un objeto, también llamada variable de instancia.

diagrama de objeto: Una representación gráfica de un objeto, sus atributos y sus valores.

especificador de formato: En una f-string, un especificador de formato determina cómo se convierte un valor en una string.

función pura: Una función que no modifica sus parámetros ni tiene ningún efecto aparte de devolver un valor.

estilo de programación funcional: Una forma de programar que usa funciones puras siempre que sea posible.

prototype and patch: Una forma de desarrollar programas empezando con un borrador aproximado y añadiendo características y corrigiendo bugs gradualmente.

design-first development: Una forma de desarrollar programas con una planificación más cuidadosa que prototype and patch.

14.11. Ejercicios

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

14.11.1. Pregunta a un asistente virtual

Hay mucho vocabulario nuevo en este capítulo. Una conversación con un asistente virtual puede ayudarte a consolidar tu comprensión. Considera preguntar:

- "¿Cuál es la diferencia entre una clase y un tipo?"
- "¿Cuál es la diferencia entre un objeto y una instancia?"
- "¿Cuál es la diferencia entre una variable y un atributo?"
- "¿Cuáles son los pros y los contras de funciones puras comparadas con funciones impuras?"

Como apenas estamos empezando con la programación orientada a objetos, el código de este capítulo no es idiomático – no es el tipo de código que escriben los programadores con experiencia. Si pides ayuda a un asistente virtual con los ejercicios, probablemente verás características que todavía no hemos cubierto. En particular, es probable que veas un método llamado `__init__` usado para inicializar los atributos de una instancia.

Si estas características tienen sentido para ti, adelante, úsalas. Pero si no, ten paciencia – llegaremos pronto. Mientras tanto, mira si puedes resolver los siguientes ejercicios usando solo las características que hemos cubierto hasta ahora.

Además, en este capítulo vimos un ejemplo de especificador de formato. Para más información, pregunta "¿Qué especificadores de formato se pueden usar en una f-string de Python?"

14.11.2. Ejercicio

Escribe una función llamada `subtract_time` que tome dos objetos `Time` y devuelva el intervalo entre ellos en segundos – asumiendo que son dos horas del mismo día.

Aquí tienes un esquema de la función para empezar.

```
def subtract_time(t1, t2):
    """Compute the difference between two times in seconds.

    >>> subtract_time(make_time(3, 2, 1), make_time(3, 2, 0))
    1
    >>> subtract_time(make_time(3, 2, 1), make_time(3, 0, 0))
    121
    >>> subtract_time(make_time(11, 12, 0), make_time(9, 40, 0))
    5520
    """
    return None
```

Puedes usar `doctest` para probar tu función.

```
from doctest import run_docstring_examples

def run_doctests(func):
    run_docstring_examples(func, globals(), name=func.__name__)

run_doctests(subtract_time)
```

14.11.3. Ejercicio

Escribe una función llamada `is_after` que tome dos objetos `Time` y devuelva `True` si la primera hora es más tarde en el día que la segunda, y `False` en caso contrario.

Aquí tienes un esquema de la función para empezar.

```
def is_after(t1, t2):
    """Checks whether `t1` is after `t2`.

    >>> is_after(make_time(3, 2, 1), make_time(3, 2, 0))
    True
    >>> is_after(make_time(3, 2, 1), make_time(3, 2, 1))
    False
    >>> is_after(make_time(11, 12, 0), make_time(9, 40, 0))
    True
    """
    return None
```

Puedes usar `doctest` para probar tu función.

```
run_doctests(is_after)
```

14.11.4. Ejercicio

Aquí tienes una definición de una clase `Date` que representa una fecha – es decir, un año, mes y día del mes.

```
class Date:
    """Represents a year, month, and day"""
```

1. Escribe una función llamada `make_date` que tome `year`, `month` y `day` como parámetros, cree un objeto `Date`, asigne los parámetros a atributos y devuelva el objeto nuevo resultante. Crea un objeto que represente el 22 de junio de 1933.
2. Escribe una función llamada `print_date` que tome un objeto `Date`, use una f-string para formatear los atributos e imprima el resultado. Si la pruebas con la `Date` que creaste, el resultado debería ser `1933-06-22`.
3. Escribe una función llamada `is_after` que tome dos objetos `Date` como parámetros y devuelva `True` si la primera viene después de la segunda. Crea un segundo objeto que represente el 17 de septiembre de 1933 y comprueba si viene después del primer objeto.

Pista: puede resultarte útil escribir una función llamada `date_to_tuple` que tome un objeto `Date` y devuelva una tupla que contenga sus atributos en orden año, mes, día.

Puedes usar este esquema de función para empezar.

```
def make_date(year, month, day):
    return None
```

Puedes usar estos ejemplos para probar `make_date`.

```
birthday1 = make_date(1933, 6, 22)
```

```
birthday2 = make_date(1933, 9, 17)
```

Puedes usar este esquema de función para empezar.

```
def print_date(date):
    print('')
```

Puedes usar este ejemplo para probar `print_date`.

```
print_date(birthday1)
```

Puedes usar este esquema de función para empezar.

```
def is_after(date1, date2):
    return None
```

Puedes usar estos ejemplos para probar `is_after`.

```
is_after(birthday1, birthday2)  # should be False
```

```
is_after(birthday2, birthday1)  # should be True
```


Traducción al español por midudev (Miguel Ángel Durán).

Puedes pedir versiones impresas y ebook de *Think Python 3e* en Bookshop.org y Amazon.

15. Clases y métodos

Python es un **lenguaje orientado a objetos** – es decir, proporciona funcionalidades que dan soporte a la programación orientada a objetos, que tiene estas características definitorias:

- La mayor parte del cálculo se expresa en términos de operaciones sobre objetos.
- Los objetos a menudo representan cosas del mundo real, y los métodos suelen corresponder a las formas en que las cosas del mundo real interactúan.
- Los programas incluyen definiciones de clase y método.

Por ejemplo, en el capítulo anterior definimos una clase `Time` que corresponde a la forma en que las personas registran la hora del día, y definimos funciones que corresponden a los tipos de cosas que las personas hacen con horas. Pero no había una conexión explícita entre la definición de la clase `Time` y las definiciones de función que vienen después. Podemos hacer explícita la conexión reescribiendo una función como un **método**, que se define dentro de una definición de clase.

15.1. Definir métodos

En el capítulo anterior definimos una clase llamada `Time` y escribimos una función llamada `print_time` que muestra una hora del día.

```
class Time:
    """Represents the time of day."""

def print_time(time):
    s = f'{time.hour:02d}:{time.minute:02d}:{time.second:02d}'
    print(s)
```

Para convertir `print_time` en un método, todo lo que tenemos que hacer es mover la definición de la función dentro de la definición de la clase. Fíjate en el cambio de indentación.

Al mismo tiempo, cambiaremos el nombre del parámetro de `time` a `self`. Este cambio no es necesario, pero es convencional que el primer parámetro de un método se llame `self`.

```
class Time:
    """Represents the time of day."""

    def print_time(self):
        s = f'{self.hour:02d}:{self.minute:02d}:{self.second:02d}'
        print(s)
```

Para llamar a este método, tienes que pasar un objeto `Time` como argumento. Aquí está la función que usaremos para crear un objeto `Time`.

```
def make_time(hour, minute, second):
    time = Time()
    time.hour = hour
    time.minute = minute
    time.second = second
    return time
```

Y aquí hay una instancia de `Time`.

```
start = make_time(9, 40, 0)
```

Ahora hay dos formas de llamar a `print_time`. La primera (y menos común) es usar sintaxis de función.

```
Time.print_time(start)
```

En esta versión, `Time` es el nombre de la clase, `print_time` es el nombre del método, y `start` se pasa como parámetro. La segunda forma (y más idiomática) es usar sintaxis de método:

```
start.print_time()
```

En esta versión, `start` es el objeto sobre el que se invoca el método, que se llama **receptor**, basándonos en la analogía de que invocar un método es como enviar un mensaje a un objeto.

Independientemente de la sintaxis, el comportamiento del método es el mismo. El receptor se asigna al primer parámetro, así que dentro del método, `self` se refiere al mismo objeto que `start`.

15.2. Otro método

Aquí está la función `time_to_int` del capítulo anterior.

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

Y aquí hay una versión reescrita como método.

```
%%add_method_to Time

    def time_to_int(self):
        minutes = self.hour * 60 + self.minute
        seconds = minutes * 60 + self.second
        return seconds
```

La primera línea usa el comando especial `add_method_to`, que añade un método a una clase definida previamente. Este comando funciona en un Jupyter notebook, pero no forma parte de Python, así que no funcionará en otros entornos. Normalmente, todos los métodos de una clase están dentro de la definición de la clase, así que se definen al mismo tiempo que la clase. Pero para este libro, es útil definir un método cada vez.

Como en el ejemplo anterior, la definición del método está indentada y el nombre del parámetro es `self`. Aparte de eso, el método es idéntico a la función. Así es como lo invocamos.

```
start.time_to_int()
```

Es común decir que "llamamos" a una función e "invocamos" un método, pero significan lo mismo.

15.3. Métodos estáticos

Como otro ejemplo, consideremos la función `int_to_time`. Aquí está la versión del capítulo anterior.

```
def int_to_time(seconds):
    minute, second = divmod(seconds, 60)
    hour, minute = divmod(minute, 60)
    return make_time(hour, minute, second)
```

Esta función toma `seconds` como parámetro y devuelve un nuevo objeto `Time`. Si la transformamos en un método de la clase `Time`, tenemos que invocarla sobre un objeto `Time`. Pero si estamos intentando crear un nuevo objeto `Time`, ¿sobre qué se supone que debemos invocarla?

Podemos resolver este problema del huevo y la gallina usando un **método estático**, que es un método que no requiere una instancia de la clase para invocarse. Así es como reescribimos esta función como un método estático.

```
%%add_method_to Time

    def int_to_time(seconds):
        minute, second = divmod(seconds, 60)
        hour, minute = divmod(minute, 60)
        return make_time(hour, minute, second)
```

Como es un método estático, no tiene `self` como parámetro. Para invocarlo, usamos `Time`, que es el objeto de clase.

```
start = Time.int_to_time(34800)
```

El resultado es un nuevo objeto que representa las 9:40.

```
start.print_time()
```

Ahora que tenemos `Time.int_to_time`, podemos usarlo para escribir `add_time` como un método. Aquí está la función del capítulo anterior.

```
def add_time(time, hours, minutes, seconds):
    duration = make_time(hours, minutes, seconds)
    seconds = time_to_int(time) + time_to_int(duration)
    return int_to_time(seconds)
```

Y aquí hay una versión reescrita como método.

```
%%add_method_to Time

    def add_time(self, hours, minutes, seconds):
        duration = make_time(hours, minutes, seconds)
        seconds = Time.time_to_int(self) + Time.time_to_int(duration)
        return Time.int_to_time(seconds)
```

`add_time` tiene `self` como parámetro porque no es un método estático. Es un método ordinario – también llamado **método de instancia**. Para invocarlo, necesitamos una instancia de `Time`.

```
end = start.add_time(1, 32, 0)
end.print_time()
```

15.4. Comparar objetos Time

Como un ejemplo más, escribamos `is_after` como método. Aquí está la función `is_after`, que es una solución a un ejercicio del capítulo anterior.

```
def is_after(t1, t2):
    return time_to_int(t1) > time_to_int(t2)
```

Y aquí está como método.

```
%%add_method_to Time

    def is_after(self, other):
        return self.time_to_int() > other.time_to_int()
```

Como estamos comparando dos objetos, y el primer parámetro es `self`, llamaremos al segundo parámetro `other`. Para usar este método, tenemos que invocarlo sobre un objeto y pasar el otro como argumento.

```
end.is_after(start)
```

Una cosa buena de esta sintaxis es que casi se lee como una pregunta: "¿`end` va después de `start`?"

15.5. El método `__str__`

Cuando escribes un método, puedes elegir casi cualquier nombre que quieras. Sin embargo, algunos nombres tienen significados especiales. Por ejemplo, si un objeto tiene un método llamado `__str__`, Python usa ese método para convertir el objeto a string. Por ejemplo, aquí hay un método `__str__` para un objeto de hora.

```
%%add_method_to Time

    def __str__(self):
        s = f'{self.hour:02d}:{self.minute:02d}:{self.second:02d}'
        return s
```

Este método es similar a `print_time`, del capítulo anterior, salvo que devuelve el string en lugar de imprimirlo.

Puedes invocar este método de la forma habitual.

```
end.__str__()
```

Pero Python también puede invocarlo por ti. Si usas la función integrada `str` para convertir un objeto `Time` a string, Python usa el método `__str__` de la clase `Time`.

```
str(end)
```

Y hace lo mismo si imprimes un objeto `Time`.

```
print(end)
```

Los métodos como `__str__` se llaman **métodos especiales**. Puedes identificarlos porque sus nombres empiezan y terminan con dos guiones bajos.

15.6. El método init

El más especial de los métodos especiales es `__init__`, llamado así porque inicializa los atributos de un nuevo objeto. Un método `__init__` para la clase `Time` podría tener este aspecto:

```
%%add_method_to Time

    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
```

Ahora, cuando instanciamos un objeto `Time`, Python invoca `__init__` y pasa los argumentos. Así podemos crear un objeto e inicializar los atributos al mismo tiempo.

```
time = Time(9, 40, 0)
print(time)
```

En este ejemplo, los parámetros son opcionales, así que si llamas a `Time` sin argumentos, obtienes los valores por defecto.

```
time = Time()
print(time)
```

Si proporcionas un argumento, sobrescribe `hour`:

```
time = Time(9)
print(time)
```

Si proporcionas dos argumentos, sobrescriben `hour` y `minute`.

```
time = Time(9, 45)
print(time)
```

Y si proporcionas tres argumentos, sobrescriben los tres valores por defecto.

Cuando escribo una clase nueva, casi siempre empiezo escribiendo `__init__`, que hace más fácil crear objetos, y `__str__`, que es útil para depurar.

15.7. Sobrecarga de operadores

Definiendo otros métodos especiales, puedes especificar el comportamiento de los operadores sobre tipos definidos por el programador. Por ejemplo, si defines un método llamado `__add__` para la clase `Time`, puedes usar el operador `+` con objetos Time.

Aquí hay un método `__add__`.

```
%%add_method_to Time

    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return Time.int_to_time(seconds)
```

Podemos usarlo así.

```
duration = Time(1, 32)
end = start + duration
print(end)
```

Están pasando muchas cosas cuando ejecutamos estas tres líneas de código:

- Cuando instanciamos un objeto `Time`, se invoca el método `__init__`.
- Cuando usamos el operador `+` con un objeto `Time`, se invoca su método `__add__`.
- Y cuando imprimimos un objeto `Time`, se invoca su método `__str__`.

Cambiar el comportamiento de un operador para que funcione con tipos definidos por el programador se llama **sobrecarga de operadores**. Para cada operador, como `+`, hay un método especial correspondiente, como `__add__`.

15.8. Depuración

Un objeto `Time` es válido si los valores de `minute` y `second` están entre `0` y `60` – incluyendo `0` pero no `60` – y si `hour` es positivo. Además, `hour` y `minute` deberían ser valores enteros, pero podríamos permitir que `second` tenga una parte fraccionaria. Requisitos como estos se llaman **invariantes** porque siempre deberían ser verdaderos. Dicho de otra forma, si no son verdaderos, algo ha ido mal.

Escribir código para comprobar invariantes puede ayudar a detectar errores y encontrar sus causas. Por ejemplo, podrías tener un método como `is_valid` que toma un objeto Time y devuelve `False` si viola una invariante.

```
%%add_method_to Time

    def is_valid(self):
        if self.hour < 0 or self.minute < 0 or self.second < 0:
            return False
        if self.minute >= 60 or self.second >= 60:
            return False
        if not isinstance(self.hour, int):
            return False
        if not isinstance(self.minute, int):
            return False
        return True
```

Entonces, al principio de cada método puedes comprobar los argumentos para asegurarte de que son válidos.

```
%%add_method_to Time

    def is_after(self, other):
        assert self.is_valid(), 'self is not a valid Time'
        assert other.is_valid(), 'other is not a valid Time'
        return self.time_to_int() > other.time_to_int()
```

La sentencia `assert` evalúa la expresión que le sigue. Si el resultado es `True`, no hace nada; si el resultado es `False`, causa un `AssertionError`. Aquí hay un ejemplo.

```
duration = Time(minute=132)
print(duration)
```

```
start.is_after(duration)
```

Las sentencias `assert` son útiles porque distinguen el código que trata con condiciones normales del código que comprueba errores.

15.9. Glosario

lenguaje orientado a objetos: Un lenguaje que proporciona funcionalidades para dar soporte a la programación orientada a objetos, especialmente tipos definidos por el usuario.

método: Una función que se define dentro de una definición de clase y se invoca sobre instancias de esa clase.

receptor (receiver): El objeto sobre el que se invoca un método.

método estático: Un método que puede invocarse sin un objeto como receptor.

método de instancia: Un método que debe invocarse con un objeto como receptor.

método especial: Un método que cambia la forma en que los operadores y algunas funciones funcionan con un objeto.

sobrecarga de operadores: El proceso de usar métodos especiales para cambiar la forma en que los operadores funcionan con tipos definidos por el usuario.

invariante: Una condición que siempre debería ser verdadera durante la ejecución de un programa.

15.10. Ejercicios

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

15.10.1. Pregunta a un asistente virtual

Para obtener más información sobre métodos estáticos, pregunta a un asistente virtual:

- "¿Cuál es la diferencia entre un método de instancia y un método estático?"
- "¿Por qué los métodos estáticos se llaman estáticos?"

Si le pides a un asistente virtual que genere un método estático, el resultado probablemente empezará con `@staticmethod`, que es un "decorator" que indica que es un método estático. Los decorators no se cubren en este libro, pero si tienes curiosidad, puedes pedirle más información a un AV.

En este capítulo reescribimos varias funciones como métodos. Los asistentes virtuales suelen ser buenos en este tipo de transformación de código. Como ejemplo, pega la siguiente función en un AV y pídele: "Reescribe esta función como un método de la clase `Time`."

```
def subtract_time(t1, t2):
    return time_to_int(t1) - time_to_int(t2)
```

15.10.2. Ejercicio

En el capítulo anterior, una serie de ejercicios te pidió escribir una clase `Date` y varias funciones que trabajan con objetos `Date`. Ahora practiquemos reescribir esas funciones como métodos.

1. Escribe una definición para una clase `Date` que represente una fecha – es decir, un año, un mes y un día del mes.
2. Escribe un método `__init__` que tome `year`, `month` y `day` como parámetros y asigne los parámetros a atributos. Crea un objeto que represente el 22 de junio de 1933.
3. Escribe un método `__str__` que use un f-string para formatear los atributos y devolver el resultado. Si lo pruebas con el `Date` que creaste, el resultado debería ser `1933-06-22`.
4. Escribe un método llamado `is_after` que tome dos objetos `Date` y devuelva `True` si el primero viene después del segundo. Crea un segundo objeto que represente el 17 de septiembre de 1933 y comprueba si viene después del primer objeto.

Pista: Podría resultarte útil escribir un método llamado `to_tuple` que devuelva una tupla que contiene los atributos de un objeto `Date` en orden año-mes-día.

Puedes usar estos ejemplos para probar tu solución.

```
birthday1 = Date(1933, 6, 22)
print(birthday1)
```

```
birthday2 = Date(1933, 9, 17)
print(birthday2)
```

```
birthday1.is_after(birthday2)  # should be False
```

```
birthday2.is_after(birthday1)  # should be True
```

Think Python: 3.ª edición

Traducción al español por midudev (Miguel Ángel Durán).

Puedes pedir versiones impresas y ebook de *Think Python 3e* en Bookshop.org y Amazon.

16. Clases y Objetos

Hasta este punto hemos definido clases y creado objetos que representan la hora del día y el día del año. Y hemos definido métodos que crean, modifican y realizan cálculos con estos objetos.

En este capítulo continuaremos nuestro recorrido por la programación orientada a objetos (OOP) definiendo clases que representan objetos geométricos, incluidos puntos, líneas, rectángulos y círculos. Escribiremos métodos que crean y modifican estos objetos, y usaremos el module `jupyturtle` para dibujarlos.

Usaré estas clases para demostrar temas de OOP, incluida la identidad y equivalencia de objetos, las copias superficiales y profundas, y el polimorfismo.

16.1. Crear un Point

En gráficos por computadora, una ubicación en la pantalla a menudo se representa usando un par de coordenadas en un plano `x`-`y`. Por convención, el punto `(0, 0)` suele representar la esquina superior izquierda de la pantalla, y `(x, y)` representa el punto que está `x` unidades a la derecha e `y` unidades hacia abajo desde el origen. Comparado con el sistema de coordenadas cartesianas que quizá hayas visto en una clase de matemáticas, el eje `y` está al revés.

Hay varias formas en que podríamos representar un punto en Python:

- Podemos almacenar las coordenadas por separado en dos variables, `x` e `y`.
- Podemos almacenar las coordenadas como elementos en una lista o tupla.
- Podemos crear un nuevo tipo para representar puntos como objetos.

En programación orientada a objetos, lo más idiomático sería crear un nuevo tipo. Para hacerlo, empezaremos con una definición de clase para `Point`.

```
class Point:
    """Represents a point in 2-D space."""

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f'Point({self.x}, {self.y})'
```

El método `__init__` toma las coordenadas como parámetros y las asigna a los atributos `x` e `y`. El método `__str__` devuelve una representación en string del `Point`.

Ahora podemos instanciar y mostrar un objeto `Point` así.

```
start = Point(0, 0)
print(start)
```

El siguiente diagrama muestra el estado del nuevo objeto.

```
from diagram import make_frame, make_binding

d1 = vars(start)
frame = make_frame(d1, name='Point', dy=-0.25, offsetx=0.18)
binding = make_binding('start', frame)
```

```
from diagram import diagram, adjust

width, height, x, y = [1.41, 0.89, 0.26, 0.5]
ax = diagram(width, height)
bbox = binding.draw(ax, x, y)
#adjust(x, y, bbox)
```

Como de costumbre, un tipo definido por el programador se representa con una caja con el nombre del tipo fuera y los atributos dentro.

En general, los tipos definidos por el programador son mutables, así que podemos escribir un método como `translate` que tome dos números, `dx` y `dy`, y los sume a los atributos `x` e `y`.

```
%%add_method_to Point

    def translate(self, dx, dy):
        self.x += dx
        self.y += dy
```

Esta función traslada el `Point` de una ubicación del plano a otra. Si no queremos modificar un `Point` existente, podemos usar `copy` para copiar el objeto original y luego modificar la copia.

```
from copy import copy

end1 = copy(start)
end1.translate(300, 0)
print(end1)
```

Podemos encapsular esos pasos en otro método llamado `translated`.

```
%%add_method_to Point

    def translated(self, dx=0, dy=0):
        point = copy(self)
        point.translate(dx, dy)
        return point
```

De la misma manera que el método `sort` modifica una lista y la función `sorted` crea una lista nueva, ahora tenemos un método `translate` que modifica un `Point` y un método `translated` que crea uno nuevo.

Aquí hay un ejemplo:

```
end2 = start.translated(0, 150)
print(end2)
```

En la siguiente sección, usaremos estos puntos para definir y dibujar una línea.

16.2. Crear una Line

Ahora definamos una clase que representa el segmento de línea entre dos puntos. Como de costumbre, empezaremos con un método `__init__` y un método `__str__`.

```
class Line:
    def __init__(self, p1, p2):
        self.p1 = p1
        self.p2 = p2

    def __str__(self):
        return f'Line({self.p1}, {self.p2})'
```

Con esos dos métodos, podemos instanciar y mostrar un objeto `Line` que usaremos para representar el eje `x`.

```
line1 = Line(start, end1)
print(line1)
```

Cuando llamamos a `print` y pasamos `line` como parámetro, `print` invoca `__str__` sobre `line`. El método `__str__` usa un f-string para crear una representación en string de la `line`.

El f-string contiene dos expresiones entre llaves, `self.p1` y `self.p2`. Cuando se evalúan esas expresiones, los resultados son objetos `Point`. Luego, cuando se convierten a strings, se invoca el método `__str__` de la clase `Point`.

Por eso, cuando mostramos una `Line`, el resultado contiene las representaciones en string de los objetos `Point`.

El siguiente diagrama de objeto muestra el estado de este objeto `Line`.

```
from diagram import Binding, Value, Frame

d1 = vars(line1.p1)
frame1 = make_frame(d1, name='Point', dy=-0.25, offsetx=0.17)

d2 = vars(line1.p2)
frame2 = make_frame(d2, name='Point', dy=-0.25, offsetx=0.17)

binding1 = Binding(Value('start'), frame1, dx=0.4)
binding2 = Binding(Value('end'), frame2, dx=0.4)
frame3 = Frame([binding1, binding2], name='Line', dy=-0.9, offsetx=0.4, offsety=-0.

binding = make_binding('line1', frame3)
```

```
width, height, x, y = [2.45, 2.12, 0.27, 1.76]
ax = diagram(width, height)
bbox = binding.draw(ax, x, y)
#adjust(x, y, bbox)
```

Las representaciones en string y los diagramas de objeto son útiles para depurar, ¡pero el objetivo de este ejemplo es generar gráficos, no texto! Así que usaremos el module `jupyturtle` para dibujar líneas en la pantalla.

Como hicimos en el Capítulo 4, usaremos `make_turtle` para crear un objeto `Turtle` y un canvas pequeño donde pueda dibujar. Para dibujar líneas, usaremos dos funciones nuevas del module `jupyturtle`:

- `jumpto`, que toma dos coordenadas y mueve la `Turtle` a la ubicación dada sin dibujar una línea, y
- `moveto`, que mueve la `Turtle` desde su ubicación actual a la ubicación dada, y dibuja un segmento de línea entre ellas.

Así es como las importamos.

```
from jupyturtle import make_turtle, jumpto, moveto
```

Y aquí hay un método que dibuja una `Line`.

```
%%add_method_to Line

    def draw(self):
        jumpto(self.p1.x, self.p1.y)
        moveto(self.p2.x, self.p2.y)
```

Para mostrar cómo se usa, crearé una segunda línea que representa el eje `y`.

```
line2 = Line(start, end2)
print(line2)
```

Y luego dibujamos los ejes.

```
make_turtle()
line1.draw()
line2.draw()
```

A medida que definamos y dibujemos más objetos, usaremos estas líneas otra vez. Pero primero hablemos de equivalencia e identidad de objetos.

16.3. Equivalencia e identidad

Supongamos que creamos dos puntos con las mismas coordenadas.

```
p1 = Point(200, 100)
p2 = Point(200, 100)
```

Si usamos el operador `==` para compararlos, obtenemos el comportamiento por defecto para tipos definidos por el programador – el resultado es `True` solo si son el mismo objeto, y no lo son.

```
p1 == p2
```

Si queremos cambiar ese comportamiento, podemos proporcionar un método especial llamado `__eq__` que define qué significa que dos objetos `Point` sean iguales.

```
%%add_method_to Point

def __eq__(self, other):
    return (self.x == other.x) and (self.y == other.y)
```

Esta definición considera que dos `Points` son iguales si sus atributos son iguales. Ahora, cuando usamos el operador `==`, invoca el método `__eq__`, que indica que `p1` y `p2` se consideran iguales.

```
p1 == p2
```

Pero el operador `is` todavía indica que son objetos diferentes.

```
p1 is p2
```

No es posible sobrescribir el operador `is` – siempre comprueba si los objetos son idénticos. Pero para tipos definidos por el programador, puedes sobrescribir el operador `==` para que compruebe si los objetos son equivalentes. Y puedes definir qué significa equivalente.

16.4. Crear un Rectangle

Ahora definamos una clase que representa y dibuja rectángulos. Para simplificar las cosas, asumiremos que los rectángulos son verticales u horizontales, no inclinados. ¿Qué atributos crees que deberíamos usar para especificar la ubicación y el tamaño de un rectángulo?

Hay al menos dos posibilidades:

- Podrías especificar el ancho y alto del rectángulo y la ubicación de una esquina.
- Podrías especificar dos esquinas opuestas.

En este punto es difícil decir si una es mejor que la otra, así que implementemos la primera. Aquí está la definición de la clase.

```
class Rectangle:
    """Represents a rectangle.

    attributes: width, height, corner.
    """
    def __init__(self, width, height, corner):
        self.width = width
        self.height = height
        self.corner = corner

    def __str__(self):
        return f'Rectangle({self.width}, {self.height}, {self.corner})'
```

Como de costumbre, el método `__init__` asigna los parámetros a atributos y `__str__` devuelve una representación en string del objeto. Ahora podemos instanciar un objeto `Rectangle`, usando un `Point` como ubicación de la esquina superior izquierda.

```
corner = Point(30, 20)
box1 = Rectangle(100, 50, corner)
print(box1)
```

El siguiente diagrama muestra el estado de este objeto.

```
from diagram import Binding, Value

def make_rectangle_binding(name, box, **options):
    d1 = vars(box.corner)
    frame_corner = make_frame(d1, name='Point', dy=-0.25, offsetx=0.07)

    d2 = dict(width=box.width, height=box.height)
    frame = make_frame(d2, name='Rectangle', dy=-0.25, offsetx=0.45)
    binding = Binding(Value('corner'), frame1, dx=0.92, draw_value=False, **options
    frame.bindings.append(binding)

    binding = Binding(Value(name), frame)
    return binding, frame_corner

binding_box1, frame_corner1 = make_rectangle_binding('box1', box1)
```

```
from diagram import Bbox

width, height, x, y = [2.83, 1.49, 0.27, 1.1]
ax = diagram(width, height)
bbox1 = binding_box1.draw(ax, x, y)
bbox2 = frame_corner1.draw(ax, x+1.85, y-0.6)
bbox = Bbox.union([bbox1, bbox2])
#adjust(x, y, bbox)
```

Para dibujar un rectángulo, usaremos el siguiente método para crear cuatro objetos `Point` que representen las esquinas.

```
%%add_method_to Rectangle

    def make_points(self):
        p1 = self.corner
        p2 = p1.translated(self.width, 0)
        p3 = p2.translated(0, self.height)
        p4 = p3.translated(-self.width, 0)
        return p1, p2, p3, p4
```

Luego crearemos cuatro objetos `Line` para representar los lados.

```
%%add_method_to Rectangle

    def make_lines(self):
        p1, p2, p3, p4 = self.make_points()
        return Line(p1, p2), Line(p2, p3), Line(p3, p4), Line(p4, p1)
```

Luego dibujaremos los lados.

```
%%add_method_to Rectangle

    def draw(self):
        lines = self.make_lines()
        for line in lines:
            line.draw()
```

Aquí hay un ejemplo.

```
make_turtle()
line1.draw()
line2.draw()
box1.draw()
```

La figura incluye dos líneas para representar los ejes.

16.5. Cambiar rectángulos

Ahora consideremos dos métodos que modifican rectángulos, `grow` y `translate`. Veremos que `grow` funciona como se esperaba, pero `translate` tiene un error sutil. A ver si puedes descubrirlo antes de que lo explique.

`grow` toma dos números, `dwidth` y `dheight`, y los suma a los atributos `width` y `height` del rectángulo.

```
%%add_method_to Rectangle

    def grow(self, dwidth, dheight):
        self.width += dwidth
        self.height += dheight
```

Aquí hay un ejemplo que demuestra el efecto creando una copia de `box1` e invocando `grow` sobre la copia.

```
box2 = copy(box1)
box2.grow(60, 40)
print(box2)
```

Si dibujamos `box1` y `box2`, podemos confirmar que `grow` funciona como se esperaba.

```
make_turtle()
line1.draw()
line2.draw()
box1.draw()
box2.draw()
```

Ahora veamos qué pasa con `translate`. Toma dos números, `dx` y `dy`, y mueve el rectángulo las distancias dadas en las direcciones `x` e `y`.

```
%%add_method_to Rectangle

    def translate(self, dx, dy):
        self.corner.translate(dx, dy)
```

Para demostrar el efecto, trasladaremos `box2` a la derecha y hacia abajo.

```
box2.translate(30, 20)
print(box2)
```

Ahora veamos qué pasa si dibujamos `box1` y `box2` otra vez.

```
make_turtle()
line1.draw()
line2.draw()
box1.draw()
box2.draw()
```

Parece que ambos rectángulos se movieron, ¡que no era lo que pretendíamos! La siguiente sección explica qué salió mal.

16.6. Copia profunda

Cuando usamos `copy` para duplicar `box1`, copia el objeto `Rectangle` pero no el objeto `Point` que contiene. Así que `box1` y `box2` son objetos diferentes, como pretendíamos.

```
box1 is box2
```

Pero sus atributos `corner` se refieren al mismo objeto.

```
box1.corner is box2.corner
```

El siguiente diagrama muestra el estado de estos objetos.

```
from diagram import Stack
from copy import deepcopy

binding_box1, frame_corner1 = make_rectangle_binding('box1', box1)
binding_box2, frame_corner2 = make_rectangle_binding('box2', box2, dy=0.4)
binding_box2.value.bindings.reverse()

stack = Stack([binding_box1, binding_box2], dy=-1.3)
```

```
from diagram import Bbox

width, height, x, y = [2.76, 2.54, 0.27, 2.16]
ax = diagram(width, height)
bbox1 = stack.draw(ax, x, y)
bbox2 = frame_corner1.draw(ax, x+1.85, y-0.6)
bbox = Bbox.union([bbox1, bbox2])
# adjust(x, y, bbox)
```

Lo que hace `copy` se llama **copia superficial** porque copia el objeto pero no los objetos que contiene. Como resultado, cambiar el `width` o `height` de un `Rectangle` no afecta al otro, ¡pero cambiar los atributos del `Point` compartido afecta a ambos! Este comportamiento es confuso y propenso a errores.

Afortunadamente, el module `copy` proporciona otra función, llamada `deepcopy`, que copia no solo el objeto sino también los objetos a los que se refiere, y los objetos a los que *ellos* se refieren, y así sucesivamente. Esta operación se llama **copia profunda**.

Para demostrarlo, empecemos con un nuevo `Rectangle` que contiene un nuevo `Point`.

```
corner = Point(20, 20)
box3 = Rectangle(100, 50, corner)
print(box3)
```

Y haremos una copia profunda.

```
from copy import deepcopy

box4 = deepcopy(box3)
```

Podemos confirmar que los dos objetos `Rectangle` se refieren a objetos `Point` diferentes.

```
box3.corner is box4.corner
```

Como `box3` y `box4` son objetos completamente separados, podemos modificar uno sin afectar al otro. Para demostrarlo, moveremos `box3` y haremos crecer `box4`.

```
box3.translate(50, 30)
box4.grow(100, 60)
```

Y podemos confirmar que el efecto es el esperado.

```
make_turtle()
line1.draw()
line2.draw()
box3.draw()
box4.draw()
```

16.7. Polimorfismo

En el ejemplo anterior, invocamos el método `draw` sobre dos objetos `Line` y dos objetos `Rectangle`. Podemos hacer lo mismo de forma más concisa creando una lista de objetos.

```
shapes = [line1, line2, box3, box4]
```

Los elementos de esta lista son de tipos diferentes, pero todos proporcionan un método `draw`, así que podemos recorrer la lista e invocar `draw` sobre cada uno.

```
make_turtle()

for shape in shapes:
    shape.draw()
```

La primera y segunda vez a través del bucle, `shape` se refiere a un objeto `Line`, así que cuando se invoca `draw`, el método que se ejecuta es el definido en la clase `Line`.

La tercera y cuarta vez a través del bucle, `shape` se refiere a un objeto `Rectangle`, así que cuando se invoca `draw`, el método que se ejecuta es el definido en la clase `Rectangle`.

En cierto sentido, cada objeto sabe cómo dibujarse a sí mismo. Esta característica se llama **polimorfismo**. La palabra viene de raíces griegas que significan "muchas formas". En programación orientada a objetos, el polimorfismo es la capacidad de diferentes tipos para proporcionar los mismos métodos, lo que hace posible realizar muchos cálculos – como dibujar formas – invocando el mismo método sobre objetos de tipos diferentes.

Como ejercicio al final de este capítulo, definirás una nueva clase que representa un círculo y proporciona un método `draw`. Luego puedes usar polimorfismo para dibujar líneas, rectángulos y círculos.

16.8. Depuración

En este capítulo, nos encontramos con un error sutil que ocurrió porque creamos un `Point` que era compartido por dos objetos `Rectangle`, y luego modificamos el `Point`. En general, hay dos formas de evitar problemas como este: puedes evitar compartir objetos o puedes evitar modificarlos.

Para evitar compartir objetos, puedes usar copia profunda, como hicimos en este capítulo.

Para evitar modificar objetos, considera reemplazar funciones impuras como `translate` por funciones puras como `translated`. Por ejemplo, aquí hay una versión de `translated` que crea un nuevo `Point` y nunca modifica sus atributos.

```
    def translated(self, dx=0, dy=0):
        x = self.x + dx
        y = self.y + dy
        return Point(x, y)
```

Python proporciona funcionalidades que hacen más fácil evitar modificar objetos. Están fuera del alcance de este libro, pero si tienes curiosidad, pregunta a un asistente virtual: "¿Cómo hago que un objeto de Python sea inmutable?"

Crear un objeto nuevo toma más tiempo que modificar uno existente, pero la diferencia rara vez importa en la práctica. Los programas que evitan objetos compartidos y funciones impuras suelen ser más fáciles de desarrollar, probar y depurar – y la mejor depuración es el que no tienes que hacer.

16.9. Glosario

copia superficial: Una operación de copia que no copia objetos anidados.

copia profunda: Una operación de copia que también copia objetos anidados.

polimorfismo: La capacidad de un método u operador para funcionar con múltiples tipos de objetos.

16.10. Ejercicios

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

16.10.1. Pregunta a un asistente virtual

Para todos los ejercicios siguientes, considera pedir ayuda a un asistente virtual. Si lo haces, querrás incluir como parte del prompt las definiciones de clase para `Point`, `Line` y `Rectangle` – de lo contrario el AV hará una suposición sobre sus atributos y funciones, y el código que genere no funcionará.

16.10.2. Ejercicio

Escribe un método `__eq__` para la clase `Line` que devuelva `True` si los objetos `Line` se refieren a objetos `Point` que son equivalentes, en cualquier orden.

Puedes usar el siguiente esquema para empezar.

```
%%add_method_to Line

def __eq__(self, other):
    return None
```

Puedes usar estos ejemplos para probar tu código.

```
start1 = Point(0, 0)
start2 = Point(0, 0)
end = Point(200, 100)
```

Este ejemplo debería ser `True` porque los objetos `Line` se refieren a objetos `Point` que son equivalentes, en el mismo orden.

```
line_a = Line(start1, end)
line_b = Line(start2, end)
line_a == line_b    # should be True
```

Este ejemplo debería ser `True` porque los objetos `Line` se refieren a objetos `Point` que son equivalentes, en orden inverso.

```
line_c = Line(end, start1)
line_a == line_c     # should be True
```

La equivalencia siempre debería ser transitiva – es decir, si `line_a` y `line_b` son equivalentes, y `line_a` y `line_c` son equivalentes, entonces `line_b` y `line_c` también deberían ser equivalentes.

```
line_b == line_c     # should be True
```

Este ejemplo debería ser `False` porque los objetos `Line` se refieren a objetos `Point` que no son equivalentes.

```
line_d = Line(start1, start2)
line_a == line_d    # should be False
```

16.10.3. Ejercicio

Escribe un método de `Line` llamado `midpoint` que calcule el punto medio de un segmento de línea y devuelva el resultado como un objeto `Point`.

Puedes usar el siguiente esquema para empezar.

```
%%add_method_to Line

    def midpoint(self):
        return Point(0, 0)
```

Puedes usar los siguientes ejemplos para probar tu código y dibujar el resultado.

```
start = Point(0, 0)
end1 = Point(300, 0)
end2 = Point(0, 150)
line1 = Line(start, end1)
line2 = Line(start, end2)
```

```
mid1 = line1.midpoint()
print(mid1)
```

```
mid2 = line2.midpoint()
print(mid2)
```

```
line3 = Line(mid1, mid2)
```

```
make_turtle()

for shape in [line1, line2, line3]:
    shape.draw()
```

16.10.4. Ejercicio

Escribe un método de `Rectangle` llamado `midpoint` que encuentre el punto en el centro de un rectángulo y devuelva el resultado como un objeto `Point`.

Puedes usar el siguiente esquema para empezar.

```
%%add_method_to Rectangle

    def midpoint(self):
        return Point(0, 0)
```

Puedes usar el siguiente ejemplo para probar tu código.

```
corner = Point(30, 20)
rectangle = Rectangle(100, 80, corner)
```

```
mid = rectangle.midpoint()
print(mid)
```

```
diagonal = Line(corner, mid)
```

```
make_turtle()

for shape in [line1, line2, rectangle, diagonal]:
    shape.draw()
```

16.10.5. Ejercicio

Escribe un método de `Rectangle` llamado `make_cross` que:

1. Use `make_lines` para obtener una lista de objetos `Line` que representan los cuatro lados del rectángulo.
2. Calcule los puntos medios de las cuatro líneas.
3. Cree y devuelva una lista de dos objetos `Line` que representan líneas que conectan puntos medios opuestos, formando una cruz por el centro del rectángulo.

Puedes usar este esquema para empezar.

```
%%add_method_to Rectangle

    def make_diagonals(self):
        return []
```

Puedes usar el siguiente ejemplo para probar tu código.

```
corner = Point(30, 20)
rectangle = Rectangle(100, 80, corner)
```

```
lines = rectangle.make_cross()
```

```
make_turtle()

rectangle.draw()
for line in lines:
    line.draw()
```

16.10.6. Ejercicio

Escribe una definición para una clase llamada `Circle` con atributos `center` y `radius`, donde `center` es un objeto Point y `radius` es un número. Incluye los métodos especiales `__init__` y `__str__`, y un método llamado `draw` que use funciones de `jupyturtle` para dibujar el círculo.

Puedes usar la siguiente función, que es una versión de la función `circle` que escribimos en el Capítulo 4.

```
from jupyturtle import make_turtle, forward, left, right
import math

def draw_circle(radius):
    circumference = 2 * math.pi * radius
    n = 30
    length = circumference / n
    angle = 360 / n
    left(angle / 2)
    for i in range(n):
        forward(length)
        left(angle)
```

Puedes usar el siguiente ejemplo para probar tu código. Empezaremos con un `Rectangle` cuadrado con `width` y `height` `100`.

```
corner = Point(20, 20)
rectangle = Rectangle(100, 100, corner)
```

El siguiente código debería crear un `Circle` que encaje dentro del cuadrado.

```
center = rectangle.midpoint()
radius = rectangle.height / 2

circle = Circle(center, radius)
print(circle)
```

Si todo funcionó correctamente, el siguiente código debería dibujar el círculo dentro del cuadrado (tocando los cuatro lados).

```
make_turtle(delay=0.01)

rectangle.draw()
circle.draw()
```

Think Python: 3.ª edición

Traducción al español por midudev (Miguel Ángel Durán).

Puedes pedir las versiones impresa y ebook de *Think Python 3e* en Bookshop.org y Amazon.

17. Herencia

La característica del lenguaje que más a menudo se asocia con la programación orientada a objetos es la **herencia**. La herencia es la capacidad de definir una nueva clase que es una versión modificada de una clase existente. En este capítulo demuestro la herencia usando clases que representan cartas de juego, barajas de cartas y manos de póker. Si no juegas al póker, no te preocupes: te diré lo que necesitas saber.

17.1. Representar cartas

Hay 52 cartas en una baraja estándar; cada una pertenece a uno de cuatro palos y a uno de trece rangos. Los palos son Spades, Hearts, Diamonds y Clubs. Los rangos son Ace, 2, 3, 4, 5, 6, 7, 8, 9,

10, Jack, Queen y King. Dependiendo del juego al que estés jugando, un Ace puede ser más alto que King o más bajo que 2.

Si queremos definir un nuevo objeto para representar una carta, es obvio cuáles deberían ser los atributos: `rank` y `suit`. Es menos obvio qué tipo deberían tener los atributos. Una posibilidad es usar strings como `'Spade'` para los palos y `'Queen'` para los rangos. Un problema con esta implementación es que no sería fácil comparar cartas para ver cuál tiene un rango o palo más alto.

Una alternativa es usar enteros para **codificar** los rangos y los palos. En este contexto, "codificar" significa que vamos a definir una correspondencia entre números y palos, o entre números y rangos. Este tipo de codificación no pretende ser secreta (eso sería "cifrado").

Por ejemplo, esta tabla muestra los palos y los códigos enteros correspondientes:

Palo	Código
Spades	3
Hearts	2
Diamonds	1
Clubs	0

Con esta codificación, podemos comparar palos comparando sus códigos.

Para codificar los rangos, usaremos el entero `2` para representar el rango `2`, `3` para representar `3`, y así sucesivamente hasta `10`. La siguiente tabla muestra los códigos para las figuras.

Rango	Código
Jack	11
Queen	12
King	13

Y podemos usar `1` o `14` para representar un Ace, dependiendo de si queremos que se considere más bajo o más alto que los demás rangos.

Para representar estas codificaciones, usaremos dos listas de strings, una con los nombres de los palos y otra con los nombres de los rangos.

Aquí tienes una definición de una clase que representa una carta, con estas listas de strings como **variables de clase**, que son variables definidas dentro de una definición de clase, pero no dentro de un método.

```
class Card:
    """Represents a standard playing card."""

    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
                  '8', '9', '10', 'Jack', 'Queen', 'King', 'Ace']
```

El primer elemento de `rank_names` es `None` porque no hay ninguna carta con rango cero. Al incluir `None` como marcador de posición, obtenemos una lista con la práctica propiedad de que el índice `2` se corresponde con el string `'2'`, y así sucesivamente.

Las variables de clase están asociadas con la clase, en lugar de con una instancia de la clase, así que podemos acceder a ellas de esta forma.

```
Card.suit_names
```

Podemos usar `suit_names` para buscar un palo y obtener el string correspondiente.

```
Card.suit_names[0]
```

Y `rank_names` para buscar un rango.

```
Card.rank_names[11]
```

17.2. Atributos de las cartas

Aquí tienes un método `__init__` para la clase `Card`: toma `suit` y `rank` como parámetros y los asigna a atributos con los mismos nombres.

```
%%add_method_to Card

    def __init__(self, suit, rank):
        self.suit = suit
        self.rank = rank
```

Ahora podemos crear un objeto `Card` así.

```
queen = Card(1, 12)
```

Podemos usar la nueva instancia para acceder a los atributos.

```
queen.suit, queen.rank
```

También es válido usar la instancia para acceder a las variables de clase.

```
queen.suit_names
```

Pero si usas la clase, queda más claro que son variables de clase, no atributos.

17.3. Imprimir cartas

Aquí tienes un método `__str__` para objetos `Card`.

```
%%add_method_to Card

    def __str__(self):
        rank_name = Card.rank_names[self.rank]
        suit_name = Card.suit_names[self.suit]
        return f'{rank_name} of {suit_name}'
```

Cuando imprimimos una `Card`, Python llama al método `__str__` para obtener una representación de la carta legible para humanos.

```
print(queen)
```

El siguiente es un diagrama del objeto de clase `Card` y de la instancia de Carta. `Card` es un objeto de clase, así que su tipo es `type`. `queen` es una instancia de `Card`, así que su tipo es `Card`. Para ahorrar espacio, no dibujé el contenido de `suit_names` y `rank_names`.

```
from diagram import Binding, Value, Frame, Stack

bindings = [Binding(Value(name), draw_value=False)
            for name in ['suit_names', 'rank_names']]

frame1 = Frame(bindings, name='type', dy=-0.5, offsetx=0.77)
binding1 = Binding(Value('Card'), frame1)

bindings = [Binding(Value(name), Value(value))
            for name, value in zip(['suit', 'rank'], [1, 11])]

frame2 = Frame(bindings, name='Card', dy=-0.3, offsetx=0.33)
binding2 = Binding(Value('queen'), frame2)

stack = Stack([binding1, binding2], dy=-1.2)
```

```
from diagram import diagram, Bbox, make_list, adjust

width, height, x, y = [2.11, 2.14, 0.35, 1.76]
ax = diagram(width, height)
bbox = stack.draw(ax, x, y)

value = make_list([])
bbox2 = value.draw(ax, x+1.66, y)

value = make_list([])
bbox3 = value.draw(ax, x+1.66, y-0.5)

bbox = Bbox.union([bbox, bbox2, bbox3])
#adjust(x, y, bbox)
```

Cada instancia de `Card` tiene sus propios atributos `suit` y `rank`, pero solo hay un objeto de clase `Card` y una sola copia de las variables de clase `suit_names` y `rank_names`.

17.4. Comparar cartas

Supongamos que creamos un segundo objeto `Card` con el mismo palo y rango.

```
queen2 = Card(1, 12)
print(queen2)
```

Si usamos el operador `==` para compararlas, comprueba si `queen` y `queen2` hacen referencia al mismo objeto.

```
queen == queen2
```

No lo hacen, así que devuelve `False`. Podemos cambiar este comportamiento definiendo el special método `__eq__`.

```
%%add_method_to Card

    def __eq__(self, other):
        return self.suit == other.suit and self.rank == other.rank
```

`__eq__` toma dos objetos `Card` como parámetros y devuelve `True` si tienen el mismo palo y rango, incluso si no son el mismo objeto. En otras palabras, comprueba si son equivalentes, aunque no sean idénticos.

Cuando usamos el operador `==` con objetos `Card`, Python llama al método `__eq__`.

```
queen == queen2
```

Como segunda prueba, vamos a crear una carta con el mismo palo y un rango diferente.

```
six = Card(1, 6)
print(six)
```

Podemos confirmar que `queen` y `six` no son equivalentes.

```
queen == six
```

Si usamos el operador `!=`, Python invoca un special método llamado `__ne__`, si existe. De lo contrario, invoca `__eq__` e invierte el resultado; así que si `__eq__` devuelve `True`, el resultado del operador `!=` es `False`.

```
queen != queen2
```

```
queen != six
```

Ahora supongamos que queremos comparar dos cartas para ver cuál es mayor. Si usamos uno de los operadores relacionales, obtenemos un `TypeError`.

```
queen < queen2
```

Para cambiar el comportamiento del operador `<`, podemos definir un special método llamado `__lt__`, abreviatura de "less than". Para este ejemplo, supongamos que el palo es más importante que el rango: por eso todos los Spades superan a todos los Hearts, que superan a todos los Diamonds, y así sucesivamente. Si dos cartas tienen el mismo palo, gana la que tiene el rango más alto.

Para implementar esta lógica, usaremos el siguiente método, que devuelve una tupla que contiene el palo y el rango de una carta, en ese orden.

```
%%add_method_to Card

    def to_tuple(self):
        return (self.suit, self.rank)
```

Podemos usar este método para escribir `__lt__`.

```
%%add_method_to Card

    def __lt__(self, other):
        return self.to_tuple() < other.to_tuple()
```

La comparación de tuplas compara los primeros elementos de cada tupla, que representan los palos. Si son iguales, compara los segundos elementos, que representan los rangos.

Ahora, si usamos el operador `<`, invoca el método `__lt__`.

```
six < queen
```

Si usamos el operador `>`, invoca un special método llamado `__gt__`, si existe. De lo contrario, invoca `__lt__` con los argumentos en el orden opuesto.

```
queen < queen2
```

```
queen > queen2
```

Por último, si usamos el operador `<=`, invoca un special método llamado `__le__`.

```
%%add_method_to Card

    def __le__(self, other):
        return self.to_tuple() <= other.to_tuple()
```

Así que podemos comprobar si una carta es menor o igual que otra.

```
queen <= queen2
```

```
queen <= six
```

Si usamos el operador `>=`, usa `__ge__` si existe. De lo contrario, invoca `__le__` con los argumentos en el orden opuesto.

```
queen >= six
```

Tal como los hemos definido, estos métodos son completos en el sentido de que podemos comparar dos objetos `Card` cualesquiera, y consistentes en el sentido de que los resultados de distintos operadores no se contradicen entre sí. Con estas dos propiedades, podemos decir que los objetos `Card` están **totalmente ordenados**. Y eso significa, como veremos pronto, que se pueden ordenar.

17.5. Barajas

Ahora que tenemos objetos que representan cartas, definamos objetos que representen barajas. La siguiente es una definición de clase para `Deck` con un método `__init__` que toma una lista de objetos `Card` como parámetro y la asigna a un atributo llamado `cards`.

```
class Deck:

    def __init__(self, cards):
        self.cards = cards
```

Para crear una lista que contiene las 52 cartas de una baraja estándar, usaremos el siguiente método estático.

```
%%add_method_to Deck

    def make_cards():
        cards = []
        for suit in range(4):
            for rank in range(2, 15):
                card = Card(suit, rank)
                cards.append(card)
        return cards
```

En `make_cards`, el bucle externo enumera los palos de `0` a `3`. El bucle interno enumera los rangos de `2` a `14`, donde `14` representa un Ace que supera a un King. Cada iteración crea una nueva `Card` con el palo y el rango actuales, y la añade a `cards`.

Así es como creamos una lista de cartas y un objeto `Deck` que la contiene.

```
cards = Deck.make_cards()
deck = Deck(cards)
len(deck.cards)
```

Contiene 52 cartas, tal como queríamos.

17.6. Imprimir la baraja

Aquí tienes un método `__str__` para `Deck`.

```
%%add_method_to Deck

    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

Este método demuestra una forma eficiente de acumular un string grande: construir una lista de strings y luego usar el método de string `join`.

Probaremos este método con una baraja que solo contiene dos cartas.

```
small_deck = Deck([queen, six])
```

Si llamamos a `str`, invoca `__str__`.

```
str(small_deck)
```

Cuando Jupyter muestra un string, enseña la forma "representacional" del string, que representa un salto de línea con la secuencia `\n`.

Sin embargo, si imprimimos el resultado, Jupyter muestra la forma "imprimible" del string, que imprime el salto de línea como espacio en blanco.

```
print(small_deck)
```

Así que las cartas aparecen en líneas separadas.

17.7. Añadir, quitar, barajar y ordenar

Para repartir cartas, nos gustaría tener un método que quite una carta de la baraja y la devuelva. El método de lista `pop` proporciona una forma cómoda de hacerlo.

```
%%add_method_to Deck

    def take_card(self):
        return self.cards.pop()
```

Así es como lo usamos.

```
card = deck.take_card()
print(card)
```

Podemos confirmar que quedan `51` cartas en la baraja.

```
len(deck.cards)
```

Para añadir una carta, podemos usar el método de lista `append`.

```
%%add_method_to Deck

    def put_card(self, card):
        self.cards.append(card)
```

Como ejemplo, podemos volver a poner la carta que acabamos de extraer.

```
deck.put_card(card)
len(deck.cards)
```

Para barajar la baraja, podemos usar la función `shuffle` del module `random`:

```
import random
```

```
# This cell initializes the random number generator so we
# always get the same results.

random.seed(3)
```

```
%%add_method_to Deck

    def shuffle(self):
        random.shuffle(self.cards)
```

Si barajamos la baraja e imprimimos las primeras cartas, podemos ver que no están en ningún orden aparente.

```
deck.shuffle()
for card in deck.cards[:4]:
    print(card)
```

Para ordenar las cartas, podemos usar el método de lista `sort`, que ordena los elementos "in place"; es decir, modifica la lista en lugar de crear una lista nueva.

```
%%add_method_to Deck

    def sort(self):
        self.cards.sort()
```

Cuando invocamos `sort`, usa el método `__lt__` para comparar cartas.

```
deck.sort()
```

Si imprimimos las primeras cartas, podemos confirmar que están en orden creciente.

```
for card in deck.cards[:4]:
    print(card)
```

En este ejemplo, `Deck.sort` no hace nada más que invocar `list.sort`. Pasar la responsabilidad de esta forma se llama **delegación**.

17.8. Padres e hijos

La herencia es la capacidad de definir una nueva clase que es una versión modificada de una clase existente. Como ejemplo, digamos que queremos una clase para representar una "mano", es decir, las cartas que tiene un jugador.

- Una mano es similar a una baraja: ambas están formadas por una colección de cartas, y ambas requieren operaciones como añadir y quitar cartas.
- Una mano también es diferente de una baraja: hay operaciones que queremos para manos que no tienen sentido para una baraja. Por ejemplo, en póker podríamos comparar dos manos para ver cuál gana. En bridge, podríamos calcular la puntuación de una mano para hacer una apuesta.

Esta relación entre clases, donde una es una versión especializada de otra, se presta a la herencia.

Para definir una nueva clase basada en una clase existente, ponemos el nombre de la clase existente entre paréntesis.

```
class Hand(Deck):
    """Represents a hand of playing cards."""
```

Esta definición indica que `Hand` hereda de `Deck`, lo que significa que los objetos `Hand` pueden acceder a métodos definidos en `Deck`, como `take_card` y `put_card`.

`Hand` también hereda `__init__` de `Deck`, pero si definimos `__init__` en la clase `Hand`, sobrescribe el de la clase `Deck`.

```
%%add_method_to Hand

    def __init__(self, label=''):
        self.label = label
        self.cards = []
```

Esta versión de `__init__` toma un string opcional como parámetro y siempre empieza con una lista vacía de cartas. Cuando creamos una `Hand`, Python invoca este método, no el de `Deck`, lo que podemos confirmar comprobando que el resultado tiene un atributo `label`.

```
hand = Hand('player 1')
hand.label
```

Para repartir una carta, podemos usar `take_card` para quitar una carta de un `Deck`, y `put_card` para añadir la carta a una `Hand`.

```
deck = Deck(cards)
card = deck.take_card()
hand.put_card(card)
print(hand)
```

Encapsulemos este código en un método de `Deck` llamado `move_cards`.

```
%%add_method_to Deck

    def move_cards(self, other, num):
        for i in range(num):
            card = self.take_card()
            other.put_card(card)
```

Este método es polimórfico; es decir, funciona con más de un tipo: `self` y `other` pueden ser tanto una `Hand` como un `Deck`. Así que podemos usar este método para repartir una carta de `Deck` a `Hand`, de una `Hand` a otra, o de una `Hand` de vuelta a un `Deck`.

Cuando una nueva clase hereda de una existente, la existente se llama **padre** y la nueva clase se llama **hija**. En general:

- Las instancias de la clase hija deberían tener todos los atributos de la clase padre, pero pueden tener atributos adicionales.
- La clase hija debería tener todos los métodos de la clase padre, pero puede tener métodos adicionales.
- Si una clase hija sobrescribe un método de la clase padre, el nuevo método debería tomar los mismos parámetros y devolver un resultado compatible.

Este conjunto de reglas se llama “principio de sustitución de Liskov”, en honor a la científica de la computación Barbara Liskov.

Si sigues estas reglas, cualquier función o método diseñado para trabajar con una instancia de una clase padre, como un `Deck`, también funcionará con instancias de una clase hija, como `Hand`. Si rompes estas reglas, tu código se derrumbará como un castillo de naipes (lo siento).

17.9. Especialización

Hagamos una clase llamada `BridgeHand` que representa una mano en bridge, un juego de cartas muy popular. Heredaremos de `Hand` y añadiremos un nuevo método llamado `high_card_point_count` que evalúa una mano usando un método de "high carta point", que suma puntos por las cartas altas de la mano.

Aquí tienes una definición de clase que contiene, como variable de clase, un diccionario que asigna nombres de cartas a sus valores de puntos.

```
class BridgeHand(Hand):
    """Represents a bridge hand."""

    hcp_dict = {
        'Ace': 4,
        'King': 3,
        'Queen': 2,
        'Jack': 1,
    }
```

Dado el rango de una carta, como `12`, podemos usar `Card.rank_names` para obtener la representación en string del rango, y luego usar `hcp_dict` para obtener su puntuación.

```
rank = 12
rank_name = Card.rank_names[rank]
score = BridgeHand.hcp_dict.get(rank_name, 0)
rank_name, score
```

El siguiente método recorre las cartas de una `BridgeHand` y suma sus puntuaciones.

```
%%add_method_to BridgeHand

    def high_card_point_count(self):
        count = 0
        for card in self.cards:
            rank_name = Card.rank_names[card.rank]
            count += BridgeHand.hcp_dict.get(rank_name, 0)
        return count
```

```
# This cell makes a fresh Deck and
# initializes the random number generator

cards = Deck.make_cards()
deck = Deck(cards)
random.seed(3)
```

Para probarlo, repartiremos una mano con cinco cartas; una mano de bridge normalmente tiene trece, pero es más fácil probar código con ejemplos pequeños.

```
hand = BridgeHand('player 2')

deck.shuffle()
deck.move_cards(hand, 5)
print(hand)
```

Y aquí está la puntuación total para el King y la Queen.

```
hand.high_card_point_count()
```

`BridgeHand` hereda las variables y métodos de `Hand` y añade una variable de clase y un método que son específicos de bridge. Esta forma de usar la herencia se llama **especialización** porque define una nueva clase especializada para un uso particular, como jugar al bridge.

17.10. Depuración

La herencia es una característica útil. Algunos programas que serían repetitivos sin herencia se pueden escribir de forma más concisa con ella. Además, la herencia puede facilitar la reutilización de código, ya que puedes personalizar el comportamiento de una clase padre sin tener que modificarla. En algunos casos, la estructura de herencia refleja la estructura natural del problema, lo que hace que el diseño sea más fácil de entender.

Por otro lado, la herencia puede hacer que los programas sean difíciles de leer. Cuando se invoca un método, a veces no está claro dónde encontrar su definición: el código relevante puede estar repartido entre varios modules.

Siempre que tengas dudas sobre el flujo de ejecución de tu programa, la solución más sencilla es añadir sentencias print al principio de los métodos relevantes. Si `Deck.shuffle` imprime un

mensaje que dice algo como `Running Deck.shuffle`, entonces, a medida que el programa se ejecuta, traza el flujo de ejecución.

Como alternativa, podrías usar la siguiente función, que toma un objeto y el nombre de un método (como string) y devuelve la clase que proporciona la definición del método.

```
def find_defining_class(obj, method_name):
    """Find the class where the given method is defined."""
    for typ in type(obj).mro():
        if method_name in vars(typ):
            return typ
    return f'Method {method_name} not found.'
```

`find_defining_class` usa el método `mro` para obtener la lista de objetos clase (tipos) que se buscarán para encontrar métodos. "MRO" significa "método resolution order", que es la secuencia de clases que Python busca para "resolver" el nombre de un método; es decir, para encontrar el objeto función al que se refiere el nombre.

Como ejemplo, vamos a instanciar una `BridgeHand` y luego encontrar la clase que define `shuffle`.

```
hand = BridgeHand('player 3')
find_defining_class(hand, 'shuffle')
```

El método `shuffle` para el objeto `BridgeHand` es el que está en `Deck`.

17.11. Glosario

herencia (herencia): La capacidad de definir una nueva clase que es una versión modificada de una clase definida previamente.

codificar (encode): Representar un conjunto de valores usando otro conjunto de valores construyendo una correspondencia entre ellos.

variable de clase: Una variable definida dentro de una definición de clase, pero no dentro de ningún método.

totalmente ordenado: Un conjunto de objetos está totalmente ordenado si podemos comparar dos elementos cualesquiera y los resultados son consistentes.

delegación: Cuando un método pasa la responsabilidad a otro método para que haga la mayor parte o todo el trabajo.

clase padre: Una clase de la que se hereda.

clase hija: Una clase que hereda de otra clase.

especialización: Una forma de usar la herencia para crear una nueva clase que es una versión especializada de una clase existente.

17.12. Ejercicios

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

17.12.1. Pregunta a un asistente virtual

Cuando sale bien, la programación orientada a objetos puede hacer que los programas sean más legibles, fáciles de probar y reutilizables. Pero también puede hacer que los programas sean complicados y difíciles de mantener. Como resultado, la OOP es un tema controvertido: algunas personas la adoran y a otras no les gusta.

Para aprender más sobre el tema, pregunta a un asistente virtual:

- ¿Cuáles son algunas ventajas y desventajas de la programación orientada a objetos?
- ¿Qué significa cuando la gente dice “favor composition over herencia”?
- ¿Qué es el principio de sustitución de Liskov?
- ¿Python es un lenguaje orientado a objetos?
- ¿Cuáles son los requisitos para que un conjunto esté totalmente ordenado?

Y, como siempre, considera usar un asistente virtual para ayudarte con los siguientes ejercicios.

17.12.2. Ejercicio

En bridge de contrato, una “trick” es una ronda de juego en la que cada uno de cuatro jugadores juega una carta. Para representar esas cartas, definiremos una clase que hereda de `Deck`.

```
class Trick(Deck):
    """Represents a trick in contract bridge."""
```

Como ejemplo, considera esta trick, donde el primer jugador empieza con el 3 of Diamonds, lo que significa que Diamonds es el “led suit”. El segundo y el tercer jugador “follow suit”, lo que significa que juegan una carta del palo que se ha iniciado. El cuarto jugador juega una carta de un palo diferente, lo que significa que no puede ganar la trick. Así que el ganador de esta trick es el tercer jugador, porque jugó la carta más alta del led suit.

```
cards = [Card(1, 3),
         Card(1, 10),
         Card(1, 12),
         Card(2, 13)]
trick = Trick(cards)
print(trick)
```

Escribe un método de `Trick` llamado `find_winner` que recorra las cartas de la `Trick` y devuelva el índice de la carta ganadora. En el ejemplo anterior, el índice de la carta ganadora es `2`.

Puedes usar el siguiente esquema para empezar.

```
%%add_method_to Trick

    def find_winner(self):
        return 0
```

Si pruebas tu método con el ejemplo anterior, el índice de la carta ganadora debería ser `2`.

```
trick.find_winner()
```

17.12.3. Ejercicio

Los siguientes ejercicios te piden escribir funciones que clasifiquen manos de póker. Si no estás familiarizado con el póker, explicaré lo que necesitas saber. Usaremos la siguiente clase para representar manos de póker.

```
class PokerHand(Hand):
    """Represents a poker hand."""

    def get_suit_counts(self):
        counter = {}
        for card in self.cards:
            key = card.suit
            counter[key] = counter.get(key, 0) + 1
        return counter

    def get_rank_counts(self):
        counter = {}
        for card in self.cards:
            key = card.rank
            counter[key] = counter.get(key, 0) + 1
        return counter
```

`PokerHand` proporciona dos métodos que ayudarán con los ejercicios.

- `get_suit_counts` recorre las cartas de la `PokerHand`, cuenta el número de cartas de cada palo y devuelve un diccionario que asigna cada código de palo al número de veces que aparece.
- `get_rank_counts` hace lo mismo con los rangos de las cartas, devolviendo un diccionario que asigna cada código de rango al número de veces que aparece.

Todos los ejercicios que siguen se pueden resolver usando solo las características de Python que hemos aprendido hasta ahora, pero algunos son más difíciles que la mayoría de los ejercicios anteriores. Te animo a pedir ayuda a un asistente virtual.

Para problemas como este, suele funcionar bien pedir consejos generales sobre estrategias y algorithms. Luego puedes escribir el código tú mismo o pedir código. Si pides código, quizá quieras proporcionar las definiciones de clase relevantes como parte del prompt.

Como primer ejercicio, escribiremos un método llamado `has_flush` que comprueba si una mano tiene un “flush”; es decir, si contiene al menos cinco cartas del mismo palo.

En la mayoría de variedades de póker, una mano contiene cinco o siete cartas, pero hay algunas variaciones exóticas donde una mano contiene otros números de cartas. Pero, independientemente de cuántas cartas haya en una mano, las únicas que cuentan son las cinco que forman la mejor mano.

Puedes usar el siguiente esquema para empezar.

```
%%add_method_to PokerHand

    def has_flush(self):
        """Checks whether this hand has a flush."""
        return False
```

Para probar este método, construiremos una mano con cinco cartas que son todas Clubs, así que contiene un flush.

```
good_hand = PokerHand('good_hand')

suit = 0
for rank in range(10, 15):
    card = Card(suit, rank)
    good_hand.put_card(card)

print(good_hand)
```

Si invocamos `get_suit_counts`, podemos confirmar que el código de rango `0` aparece `5` veces.

```
good_hand.get_suit_counts()
```

Así que `has_flush` debería devolver `True`.

```
good_hand.has_flush()
```

Como segunda prueba, construiremos una mano con tres Clubs y otros dos palos.

```
cards = [Card(0, 2),
         Card(0, 3),
         Card(2, 4),
         Card(3, 5),
         Card(0, 7),
        ]

bad_hand = PokerHand('bad hand')
for card in cards:
    bad_hand.put_card(card)

print(bad_hand)
```

Así que `has_flush` debería devolver `False`.

```
bad_hand.has_flush()
```

17.12.4. Ejercicio

Escribe un método llamado `has_straight` que compruebe si una mano contiene una straight, que es un conjunto de cinco cartas con rangos consecutivos. Por ejemplo, si una mano contiene los rangos `5`, `6`, `7`, `8` y `9`, contiene una straight.

Un Ace puede ir antes de un dos o después de un King, así que `Ace`, `2`, `3`, `4`, `5` es una straight, y también lo es `10`, `Jack`, `Queen`, `King`, `Ace`. Pero una straight no puede "dar la vuelta", así que `King`, `Ace`, `2`, `3`, `4` no es una straight.

Puedes usar el siguiente esquema para empezar. Incluye unas pocas líneas de código que cuentan el número de Aces, representados con el código `1` o `14`, y almacenan el total en ambas ubicaciones del contador.

```
%%add_method_to PokerHand

    def has_straight(self, n=5):
        """Checks whether this hand has a straight with at least `n` cards."""
        counter = self.get_rank_counts()
        aces = counter.get(1, 0) + counter.get(14, 0)
        counter[1] = aces
        counter[14] = aces

        return False
```

`good_hand`, que creamos para el ejercicio anterior, contiene una straight. Si usamos `get_rank_counts`, podemos confirmar que tiene al menos una carta de cada uno de cinco rangos consecutivos.

```
good_hand.get_rank_counts()
```

Así que `has_straight` debería devolver `True`.

```
good_hand.has_straight()
```

`bad_hand` no contiene una straight, así que `has_straight` debería devolver `False`.

```
bad_hand.has_straight()
```

17.12.5. Ejercicio

Una mano tiene una straight flush si contiene un conjunto de cinco cartas que son a la vez una straight y un flush; es decir, cinco cartas del mismo palo con rangos consecutivos. Escribe un método de `PokerHand` que compruebe si una mano tiene una straight flush.

Puedes usar el siguiente esquema para empezar.

```
%%add_method_to PokerHand

    def has_straightflush(self):
        """Check whether this hand has a straight flush."""
        return False
```

Usa los siguientes ejemplos para probar tu método.

```
good_hand.has_straightflush()     # should return True
```

```
bad_hand.has_straightflush()     # should return False
```

Ten en cuenta que no basta con comprobar si una mano tiene una straight y un flush. Para ver por qué, considera la siguiente mano.

```
from copy import deepcopy

straight_and_flush = deepcopy(bad_hand)
straight_and_flush.put_card(Card(0, 6))
straight_and_flush.put_card(Card(0, 9))
print(straight_and_flush)
```

Esta mano contiene una straight y un flush, pero no son las mismas cinco cartas.

```
 straight_and_flush.has_straight(), straight_and_flush.has_flush()
```

Así que no contiene una straight flush.

```
straight_and_flush.has_straightflush()    # should return False
```

17.12.6. Ejercicio

Una mano de póker tiene una pareja si contiene dos o más cartas con el mismo rango. Escribe un método de `PokerHand` que compruebe si una mano contiene una pareja.

Puedes usar el siguiente esquema para empezar.

```
%%add_method_to PokerHand

    def check_sets(self, *need_list):
        return True
```

Para probar tu método, aquí tienes una mano que tiene una pareja.

```
pair = deepcopy(bad_hand)
pair.put_card(Card(1, 2))
print(pair)
```

```
pair.has_pair()    # should return True
```

```
bad_hand.has_pair()    # should return False
```

```
good_hand.has_pair()   # should return False
```

17.12.7. Ejercicio

Una mano tiene un full house si contiene tres cartas de un rango y dos cartas de otro rango. Escribe un método de `PokerHand` que compruebe si una mano tiene un full house.

Puedes usar el siguiente esquema para empezar.

```
%%add_method_to PokerHand

    def has_full_house(self):
        return False
```

Puedes usar esta mano para probar tu método.

```
boat = deepcopy(pair)
boat.put_card(Card(2, 2))
boat.put_card(Card(2, 3))
print(boat)
```

```
boat.has_full_house()     # should return True
```

```
pair.has_full_house()     # should return False
```

```
good_hand.has_full_house()     # should return False
```

17.12.8. Ejercicio

Este ejercicio es una advertencia sobre un error común que puede ser difícil de depurar. Considera la siguiente definición de clase.

```
class Kangaroo:
    """A Kangaroo is a marsupial."""

    def __init__(self, name, contents=[]):
        """Initialize the pouch contents.

        name: string
        contents: initial pouch contents.
        """
        self.name = name
        self.contents = contents

    def __str__(self):
        """Return a string representaion of this Kangaroo.
        """
        t = [ self.name + ' has pouch contents:' ]
        for obj in self.contents:
            s = '    ' + object.__str__(obj)
            t.append(s)
        return '\n'.join(t)

    def put_in_pouch(self, item):
        """Adds a new item to the pouch contents.

        item: object to be added
        """
        self.contents.append(item)
```

`__init__` toma dos parámetros: `name` es obligatorio, pero `contents` es opcional; si no se proporciona, el valor por defecto es una lista vacía.

`__str__` devuelve una representación en string del objeto que incluye el nombre y el contenido de la bolsa.

`put_in_pouch` toma cualquier objeto y lo añade a `contents`.

Ahora veamos cómo funciona esta clase. Crearemos dos objetos `Kangaroo` con los nombres `'Kanga'` y `'Roo'`.

```
kanga = Kangaroo('Kanga')
roo = Kangaroo('Roo')
```

A la bolsa de Kanga le añadiremos dos strings y Roo.

```
kanga.put_in_pouch('wallet')
kanga.put_in_pouch('car keys')
kanga.put_in_pouch(roo)
```

Si imprimimos `kanga`, parece que todo ha funcionado.

```
print(kanga)
```

Pero ¿qué pasa si imprimimos `roo`?

```
print(roo)
```

¡La bolsa de Roo contiene el mismo contenido que la de Kanga, incluida una referencia a `roo`!

A ver si puedes averiguar qué salió mal. Luego pregunta a un asistente virtual: "¿Qué está mal en el siguiente programa?" y pega la definición de `Kangaroo`.

Think Python: 3.ª edición

Traducción al español por midudev (Miguel Ángel Durán).

Puedes comprar las versiones impresa y ebook de *Think Python 3e* en Bookshop.org y Amazon.

Aquí tienes versiones de las clases `Card`, `Deck` y `Hand` del Capítulo 17, que usaremos en algunos ejemplos de este capítulo.

```
class Card:
    suit_names = ['Clubs', 'Diamonds', 'Hearts', 'Spades']
    rank_names = [None, 'Ace', '2', '3', '4', '5', '6', '7',
                  '8', '9', '10', 'Jack', 'Queen', 'King', 'Ace']

    def __init__(self, suit, rank):
        self.suit = suit
        self.rank = rank

    def __str__(self):
        rank_name = Card.rank_names[self.rank]
        suit_name = Card.suit_names[self.suit]
        return f'{rank_name} of {suit_name}'
```

```
import random

class Deck:
    def __init__(self, cards):
        self.cards = cards

    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)

    def make_cards():
        cards = []
        for suit in range(4):
            for rank in range(2, 15):
                card = Card(suit, rank)
                cards.append(card)
        return cards

    def shuffle(self):
        random.shuffle(self.cards)

    def pop_card(self):
        return self.cards.pop()

    def add_card(self, card):
        self.cards.append(card)
```

```
class Hand(Deck):
    def __init__(self, label=''):
        self.label = label
        self.cards = []
```

18. Extras de Python

Uno de mis objetivos con este libro ha sido enseñarte la menor cantidad posible de Python. Cuando había dos maneras de hacer algo, elegí una y evité mencionar la otra. O a veces puse la segunda en un ejercicio.

Ahora quiero volver a algunas de las partes buenas que quedaron fuera. Python ofrece varias características que no son realmente necesarias – puedes escribir buen código sin ellas – pero con ellas puedes escribir código más conciso, legible o eficiente, y a veces las tres cosas.

18.1. Conjuntos

Python proporciona una clase llamada `set` que representa una colección de elementos únicos. Para crear un conjunto vacío, podemos usar el objeto de clase como una función.

```
s1 = set()
s1
```

Podemos usar el método `add` para añadir elementos.

```
s1.add('a')
s1.add('b')
s1
```

O podemos pasar cualquier tipo de secuencia a `set`.

```
s2 = set('acd')
s2
```

Un elemento solo puede aparecer una vez en un `set`. Si añades un elemento que ya está, no tiene ningún efecto.

```
s1.add('a')
s1
```

O, si creas un conjunto con una secuencia que contiene duplicados, el resultado contiene solo elementos únicos.

```
set('banana')
```

Algunos de los ejercicios de este libro se pueden hacer de forma concisa y eficiente con conjuntos. Por ejemplo, aquí tienes una solución a un ejercicio del Capítulo 11 que usa un diccionario para comprobar si hay elementos duplicados en una secuencia.

```
def has_duplicates(t):
    d = {}
    for x in t:
        d[x] = True
    return len(d) < len(t)
```

Esta versión añade los elementos de `t` como claves en un diccionario, y luego comprueba si hay menos claves que elementos. Usando conjuntos, podemos escribir la misma función así.

```
def has_duplicates(t):
    s = set(t)
    return len(s) < len(t)
```

```
has_duplicates('abba')
```

Un elemento solo puede aparecer en un conjunto una vez, así que si un elemento de `t` aparece más de una vez, el conjunto será más pequeño que `t`. Si no hay duplicados, el conjunto tendrá el mismo tamaño que `t`.

Los objeto `set` proporcionan métodos que realizan operaciones de conjuntos. Por ejemplo, `union` calcula la unión de dos conjuntos, que es un nuevo conjunto que contiene todos los elementos que aparecen en cualquiera de los dos conjuntos.

```
s1.union(s2)
```

Algunos operadores aritméticos funcionan con conjuntos. Por ejemplo, el operador `-` realiza la resta de conjuntos – el resultado es un nuevo conjunto que contiene todos los elementos del primer conjunto que *no* están en el segundo conjunto.

```
s1 - s2
```

En el Capítulo 12 usamos diccionarios para encontrar las palabras que aparecen en un documento pero no en una lista de palabras. Usamos la siguiente función, que recibe dos diccionarios y devuelve un nuevo diccionario que contiene solo las claves del primero que no aparecen en el segundo.

```
def subtract(d1, d2):
    res = {}
    for key in d1:
        if key not in d2:
            res[key] = d1[key]
    return res
```

Con conjuntos, no tenemos que escribir esta función nosotros mismos. Si `word_counter` es un diccionario que contiene las palabras únicas del documento y `word_list` es una lista de palabras válidas, podemos calcular la diferencia de conjuntos así.

```
# this cell creates a small example so we can run the following
# cell without loading the actual data

word_counter = {'word': 1}
word_list = ['word']
```

```
set(word_counter) - set(word_list)
```

El resultado es un conjunto que contiene las palabras del documento que no aparecen en la lista de palabras.

Los operadores relacionales funcionan con conjuntos. Por ejemplo, `<=` comprueba si un conjunto es un subconjunto de otro, incluyendo la posibilidad de que sean iguales.

```
set('ab') <= set('abc')
```

Con estos operadores, podemos usar conjuntos para hacer algunos de los ejercicios del Capítulo 7. Por ejemplo, aquí tienes una versión de `uses_only` que usa un bucle.

```
def uses_only(word, available):
    for letter in word:
        if letter not in available:
            return False
    return True
```

`uses_only` comprueba si todas las letras de `word` están en `available`. Con conjuntos, podemos reescribirla así.

```
def uses_only(word, available):
    return set(word) <= set(available)
```

Si las letras de `word` son un subconjunto de las letras de `available`, eso significa que `word` usa solo letras de `available`.

18.2. Counters

Un `Counter` es como un conjunto, excepto que si un elemento aparece más de una vez, el `Counter` lleva la cuenta de cuántas veces aparece. Si conoces la idea matemática de un "multiset", un `Counter` es una forma natural de representar un multiset.

La clase `Counter` está definida en un module llamado `collections`, así que tienes que importarla. Después puedes usar el objeto de clase como una función y pasar como argumento un string, una lista o cualquier otro tipo de secuencia.

```
from collections import Counter

counter = Counter('banana')
counter
```

```
from collections import Counter

t = (1, 1, 1, 2, 2, 3)
counter = Counter(t)
counter
```

Un objeto `Counter` es como un diccionario que asocia cada clave con el número de veces que aparece. Como en los diccionarios, las claves tienen que ser hashable.

A diferencia de los diccionarios, los objeto `Counter` no lanzan una excepción si accedes a un elemento que no aparece. En su lugar, devuelven `0`.

```
counter['d']
```

Podemos usar objeto `Counter` para resolver uno de los ejercicios del Capítulo 10, que pide una función que reciba dos palabras y compruebe si son anagramas – es decir, si las letras de una se pueden reordenar para formar la otra.

Aquí tienes una solución usando objeto `Counter`.

```
def is_anagram(word1, word2):
    return Counter(word1) == Counter(word2)
```

Si dos palabras son anagramas, contienen las mismas letras con los mismos recuentos, así que sus objeto `Counter` son equivalentes.

`Counter` proporciona un método llamado `most_common` que devuelve una lista de pares valor-frecuencia, ordenados de más común a menos común.

```
counter.most_common()
```

También proporcionan métodos y operadores para realizar operaciones parecidas a las de conjuntos, incluyendo suma, resta, unión e intersección. Por ejemplo, el operador `+` combina dos objeto `Counter` y crea un nuevo `Counter` que contiene las claves de ambos y las sumas de los recuentos.

Podemos probarlo creando un `Counter` con las letras de `'bans'` y sumándolo a las letras de `'banana'`.

```
counter2 = Counter('bans')
counter + counter2
```

Tendrás la oportunidad de explorar otras operaciones de `Counter` en los ejercicios al final de este capítulo.

18.3. defaultdict

El module `collections` también proporciona `defaultdict`, que es como un diccionario excepto que, si accedes a una clave que no existe, genera automáticamente un nuevo valor.

Cuando creas un `defaultdict`, proporcionas una función que se usa para crear nuevos valores. Una función que crea objetos a veces se llama **fábrica**. Las funciones integradas que crean listas, conjuntos y otros tipos se pueden usar como fábricas.

Por ejemplo, aquí tienes un `defaultdict` que crea una nueva `list` cuando hace falta.

```
from collections import defaultdict

d = defaultdict(list)
d
```

Fíjate en que el argumento es `list`, que es un objeto de clase, no `list()`, que es una llamada a función que crea una nueva lista. La fábrica función no se llama a menos que accedamos a una clave que no existe.

```
t = d['new key']
t
```

La nueva lista, que llamamos `t`, también se añade al diccionario. Así que si modificamos `t`, el cambio aparece en `d`:

```
t.append('new value')
d['new key']
```

Si estás creando un diccionario de listas, a menudo puedes escribir código más simple usando `defaultdict`.

En uno de los ejercicios del Capítulo 11, hice un diccionario que asocia un string de letras ordenadas con la lista de palabras que se pueden formar con esas letras. Por ejemplo, el string `'opst'` se asocia con la lista `['opts', 'post', 'pots', 'spot', 'stop', 'tops']`. Aquí está el código original.

```
def all_anagrams(filename):
    d = {}
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        if t not in d:
            d[t] = [word]
        else:
            d[t].append(word)
    return d
```

Y aquí tienes una versión más simple usando un `defaultdict`.

```
def all_anagrams(filename):
    d = defaultdict(list)
    for line in open(filename):
        word = line.strip().lower()
        t = signature(word)
        d[t].append(word)
    return d
```

En los ejercicios al final del capítulo, tendrás la oportunidad de practicar usando objeto `defaultdict`.

```
from collections import defaultdict

d = defaultdict(list)
key = ('into', 'the')
d[key].append('woods')
d[key]
```

18.4. Expresiones condicionales

Las sentencias condicionales se usan a menudo para elegir uno de dos valores, como aquí:

```
import math
x = 5
```

```
if x > 0:
    y = math.log(x)
else:
    y = float('nan')
```

```
y
```

Esta sentencia comprueba si `x` es positivo. Si lo es, calcula su logaritmo. Si no, `math.log` lanzaría un ValueError. Para evitar detener el programa, generamos un `NaN`, que es un valor especial de punto flotante que representa "Not a Number".

Podemos escribir esta sentencia de forma más concisa usando una **conditional expression**.

```
y = math.log(x) if x > 0 else float('nan')
```

```
y
```

Casi puedes leer esta línea como si fuera inglés: "`y` recibe log-`x` si `x` es mayor que 0; de lo contrario recibe `NaN`".

Las funciones recursivas a veces se pueden escribir de forma concisa usando expresiones condicionales. Por ejemplo, aquí tienes una versión de `factorial` con una *sentencia* condicional.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Y aquí tienes una versión con una *expresión* condicional.

```
def factorial(n):
    return 1 if n == 0 else n * factorial(n-1)
```

Otro uso de las expresiones condicionales es manejar argumentos opcionales. Por ejemplo, aquí tienes una definición de clase con un método `__init__` que usa una sentencia condicional para comprobar un parámetro con un valor por defecto.

```
class Kangaroo:
    def __init__(self, name, contents=None):
        self.name = name
        if contents is None:
            contents = []
        self.contents = contents
```

Aquí tienes una versión que usa una expresión condicional.

```
def __init__(self, name, contents=None):
    self.name = name
    self.contents = [] if contents is None else contents
```

En general, puedes reemplazar una sentencia condicional por una expresión condicional si ambas ramas contienen una sola expresión y ninguna sentencia.

18.5. Comprensiones de listas

En capítulos anteriores, hemos visto algunos ejemplos donde empezamos con una lista vacía y añadimos elementos, uno por uno, usando el método `append`. Por ejemplo, supón que tenemos un string que contiene el título de una película, y queremos poner en mayúscula la primera letra de todas las palabras.

```
title = 'monty python and the holy grail'
```

Podemos dividirlo en una lista de strings, recorrer los strings, capitalizarlos y añadirlos a una lista.

```
t = []
for word in title.split():
    t.append(word.capitalize())

' '.join(t)
```

Podemos hacer lo mismo de forma más concisa usando una **comprensión de listas**:

```
t = [word.capitalize() for word in title.split()]

' '.join(t)
```

Los operadores de corchetes indican que estamos construyendo una nueva lista. La expresión dentro de los corchetes especifica los elementos de la lista, y la cláusula `for` indica qué secuencia estamos recorriendo.

La sintaxis de una comprensión de listas puede parecer extraña, porque la variable del bucle – `word` en este ejemplo – aparece en la expresión antes de llegar a su definición. Pero te acostumbras.

Como otro ejemplo, en el Capítulo 9 usamos este bucle para leer palabras de un archivo y añadirlas a una lista.

```
download('https://raw.githubusercontent.com/AllenDowney/ThinkPython2/master/code/wo
```

```
word_list = []

for line in open('words.txt'):
    word = line.strip()
    word_list.append(word)
```

```
len(word_list)
```

Así es como podemos escribirlo como una comprensión de listas.

```
word_list = [line.strip() for line in open('words.txt')]
```

```
len(word_list)
```

Una comprensión de listas también puede tener una cláusula `if` que determina qué elementos se incluyen en la lista. Por ejemplo, aquí tienes un bucle `for` que usamos en el Capítulo 10 para crear una lista solo con las palabras de `word_list` que son palíndromos.

```
def is_palindrome(word):
    return list(reversed(word)) == list(word)
```

```
palindromes = []

for word in word_list:
    if is_palindrome(word):
        palindromes.append(word)
```

```
palindromes[:10]
```

Así es como podemos hacer lo mismo con una comprensión de listas.

```
palindromes = [word for word in word_list if is_palindrome(word)]
```

```
palindromes[:10]
```

Cuando una comprensión de listas se usa como argumento de una función, a menudo podemos omitir los corchetes. Por ejemplo, supón que queremos sumar $1/2^n$ para valores de n de 0 a 9. Podemos usar una comprensión de listas así.

```
sum([1/2**n for n in range(10)])
```

O podemos dejar fuera los corchetes así.

```
sum(1/2**n for n in range(10))
```

En este ejemplo, técnicamente el argumento es una **expresión generadora**, no una comprensión de listas, y en realidad nunca crea una lista. Pero aparte de eso, el comportamiento es el mismo.

Las comprensiones de listas y las expresiones generadoras son concisas y fáciles de leer, al menos para expresiones simples. Y normalmente son más rápidas que los bucles `for` equivalentes, a veces mucho más rápidas. Así que si estás enfadado conmigo por no haberlas mencionado antes, lo entiendo.

Pero, en mi defensa, las comprensiones de listas son más difíciles de depurar porque no puedes poner una sentencia print dentro del bucle. Te sugiero que las uses solo si el cálculo es lo bastante

simple como para que probablemente te salga bien a la primera. O considera escribir y depurar un bucle `for` y luego convertirlo en una comprensión de listas.

18.6. `any` y `all`

Python proporciona una función integrada, `any`, que recibe una secuencia de valores booleanos y devuelve `True` si cualquiera de los valores es `True`.

```
any([False, False, True])
```

`any` se usa a menudo con expresiones generadoras.

```
any(letter == 't' for letter in 'monty')
```

Ese ejemplo no es muy útil porque hace lo mismo que el operador `in`. Pero podríamos usar `any` para escribir soluciones concisas a algunos de los ejercicios del Capítulo 7. Por ejemplo, podemos escribir `uses_none` así.

```
def uses_none(word, forbidden):
    """Checks whether a word avoids forbidden letters."""
    return not any(letter in forbidden for letter in word)
```

```
uses_none('banana', 'xyz')
```

```
uses_none('apple', 'efg')
```

Esta función recorre las letras de `word` y comprueba si alguna de ellas está en `forbidden`. Usar `any` con una expresión generadora es eficiente porque se detiene inmediatamente si encuentra un valor `True`, así que no tiene que recorrer toda la secuencia.

Python proporciona otra función integrada, `all`, que devuelve `True` si todos los elementos de la secuencia son `True`. Podemos usarla para escribir una versión concisa de `uses_all`.

```
def uses_all(word, required):
    """Check whether a word uses all required letters."""
    return all(letter in word for letter in required)
```

```
uses_all('banana', 'ban')
```

```
uses_all('apple', 'api')
```

Las expresiones que usan `any` y `all` pueden ser concisas, eficientes y fáciles de leer.

18.7. Named tuplas

El module `collections` proporciona una función llamada `namedtuple` que se puede usar para crear clases simples. Por ejemplo, el objeto `Point` del Capítulo 16 solo tiene dos atributos, `x` e `y`. Así es como lo definimos.

```
class Point:
    """Represents a point in 2-D space."""

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f'({self.x}, {self.y})'
```

Eso es mucho código para transmitir una pequeña cantidad de información. `namedtuple` proporciona una forma más concisa de definir clases como esta.

```
from collections import namedtuple

Point = namedtuple('Point', ['x', 'y'])
```

El primer argumento es el nombre de la clase que quieres crear. El segundo es una lista de los atributos que deben tener los objeto `Point`. El resultado es un objeto de clase, por eso se asigna a un nombre de variable con mayúscula inicial.

Una clase creada con `namedtuple` proporciona un método `__init__` que asigna valores a los atributos y un `__str__` que muestra el objeto de forma legible. Así que podemos crear y mostrar un objeto `Point` así.

```
p = Point(1, 2)
p
```

`Point` también proporciona un método `__eq__` que comprueba si dos objeto `Point` son equivalentes – es decir, si sus atributos son iguales.

```
p == Point(1, 2)
```

Puedes acceder a los elementos de una named tupla por nombre o por índice.

```
p.x, p.y
```

```
p[0], p[1]
```

También puedes tratar una named tupla como una tupla, como en esta asignación.

```
x, y = p
x, y
```

Pero los objeto `namedtuple` son inmutables. Después de inicializar los atributos, no se pueden cambiar.

```
p[0] = 3
```

```
p.x = 3
```

`namedtuple` proporciona una forma rápida de definir clases simples. La desventaja es que las clases simples no siempre siguen siendo simples. Puede que más adelante decidas que quieres añadir métodos a una named tupla. En ese caso, puedes definir una nueva clase que herede de la named tupla.

```
class Pointier(Point):
    """This class inherits from Point"""
```

O en ese punto podrías cambiar a una definición de clase convencional.

18.8. Empaquetar argumentos de palabra clave

En el Capítulo 11, escribimos una función que empaqueta sus argumentos en una tupla.

```
def mean(*args):
    return sum(args) / len(args)
```

Puedes llamar a esta función con cualquier número de argumentos.

```
mean(1, 2, 3)
```

Pero el operador `*` no empaqueta argumentos de palabra clave. Así que llamar a esta función con un argumento de palabra clave causa un error.

```
mean(1, 2, start=3)
```

Para empaquetar argumentos de palabra clave, podemos usar el operador `**`:

```
def mean(*args, **kwargs):
    print(kwargs)
    return sum(args) / len(args)
```

El parámetro que empaqueta keywords puede tener cualquier nombre, pero `kwargs` es una elección común. El resultado es un diccionario que asocia keywords con valores.

```
mean(1, 2, start=3)
```

En este ejemplo, se imprime el valor de `kwargs`, pero por lo demás no tiene ningún efecto.

Pero el operador `**` también se puede usar en una lista de argumentos para desempaquetar un diccionario. Por ejemplo, aquí tienes una versión de `mean` que empaqueta cualquier argumento de palabra clave que recibe y luego los desempaqueta como argumentos de palabra clave para `sum`.

```
def mean(*args, **kwargs):
    return sum(args, **kwargs) / len(args)
```

Ahora, si llamamos a `mean` con `start` como argumento de palabra clave, se pasa a sum, que lo usa como punto de partida de la suma. En el siguiente ejemplo `start=3` añade `3` a la suma antes de calcular la media, así que la suma es `6` y el resultado es `3`.

```
mean(1, 2, start=3)
```

Como otro ejemplo, si tenemos un diccionario con las claves `x` e `y`, podemos usarlo con el operador de desempaquetado para crear un objeto `Point`.

```
d = dict(x=1, y=2)
Point(**d)
```

Sin el operador de desempaquetado, `d` se trata como un único argumento posicional, así que se asigna a `x`, y obtenemos un `TypeError` porque no hay un segundo argumento que asignar a `y`.

```
d = dict(x=1, y=2)
Point(d)
```

Cuando trabajas con funciones que tienen un gran número de argumentos de palabra clave, a menudo es útil crear y pasar diccionarios que especifican opciones usadas con frecuencia.

```
def pack_and_print(**kwargs):
    print(kwargs)

pack_and_print(a=1, b=2)
```

18.9. Depuración

En capítulos anteriores, usamos `doctest` para probar funciones. Por ejemplo, aquí tienes una función llamada `add` que recibe dos números y devuelve su suma. Incluye un doctest que comprueba si `2 + 2` es `4`.

```
def add(a, b):
    '''Add two numbers.

    >>> add(2, 2)
    4
    '''
    return a + b
```

Esta función recibe un objeto función y ejecuta sus doctests.

```
from doctest import run_docstring_examples

def run_doctests(func):
    run_docstring_examples(func, globals(), name=func.__name__)
```

Así que podemos probar `add` así.

```
run_doctests(add)
```

No hay salida, lo que significa que todas las pruebas pasaron.

Python proporciona otra herramienta para ejecutar pruebas automatizadas, llamada `unittest`. Es un poco más complicada de usar, pero aquí tienes un ejemplo.

```
from unittest import TestCase

class TestExample(TestCase):

    def test_add(self):
        result = add(2, 2)
        self.assertEqual(result, 4)
```

Primero importamos `TestCase`, que es una clase del module `unittest`. Para usarla, tenemos que definir una nueva clase que herede de `TestCase` y proporcione al menos un método de test. El

nombre del método de test debe empezar con `test` y debería indicar qué función prueba.

En este ejemplo, `test_add` prueba la función `add` llamándola, guardando el resultado e invocando `assertEqual`, que se hereda de `TestCase`. `assertEqual` recibe dos argumentos y comprueba si son iguales.

Para ejecutar este método de test, tenemos que ejecutar una función de `unittest` llamada `main` y proporcionar varios argumentos de palabra clave. La siguiente función muestra los detalles – si tienes curiosidad, puedes pedirle a un asistente virtual que explique cómo funciona.

```
import unittest

def run_unittest():
    unittest.main(argv=[''], verbosity=0, exit=False)
```

`run_unittest` no recibe `TestExample` como argumento – en su lugar, busca clases que heredan de `TestCase`. Luego busca métodos que empiezan con `test` y los ejecuta. Este proceso se llama **test discovery**.

Esto es lo que pasa cuando llamamos a `run_unittest`.

```
run_unittest()
```

`unittest.main` informa del número de pruebas que ejecutó y de los resultados. En este caso `OK` indica que las pruebas pasaron.

Para ver qué pasa cuando un test falla, añadiremos un método de test incorrecto a `TestExample`.

```
%%add_method_to TestExample

    def test_add_broken(self):
        result = add(2, 2)
        self.assertEqual(result, 100)
```

Esto es lo que pasa cuando ejecutamos las pruebas.

```
run_unittest()
```

El informe incluye el método de test que falló y un mensaje de error que muestra dónde. El resumen indica que se ejecutaron dos pruebas y una falló.

En los ejercicios siguientes, sugeriré algunos prompts que puedes usar para pedir más información sobre `unittest` a un asistente virtual.

18.10. Glosario

fábrica: Una función usada para crear objetos, que a menudo se pasa como parámetro a una función.

conditional expression: Una expresión que usa un condicional para seleccionar uno de dos valores.

comprensión de listas: Una forma concisa de recorrer una secuencia y crear una lista.

expresión generadora: Similar a una comprensión de listas, excepto que no crea una lista.

test discovery: Un proceso usado para encontrar y ejecutar pruebas.

18.11. Ejercicios

```
# This cell tells Jupyter to provide detailed debugging information
# when a runtime error occurs. Run it before working on the exercises.

%xmode Verbose
```

18.11.1. Pregunta a un asistente virtual

Hay algunos temas de este capítulo sobre los que quizá quieras aprender más.

- "¿Cuáles son los métodos y operadores de la clase conjunto de Python?"
- "¿Cuáles son los métodos y operadores de la clase Counter de Python?"
- "¿Cuál es la diferencia entre una comprensión de listas de Python y una expresión generadora?"
- "¿Cuándo debería usar `namedtuple` de Python en lugar de definir una nueva clase?"

- "¿Cuáles son algunos usos de empaquetar y desempaquetar argumentos de palabra clave?"
- "¿Cómo hace `unittest` test discovery?"
- "Junto con `assertEqual`, ¿cuáles son los métodos más usados en `unittest.TestCase`?"
- "¿Cuáles son los pros y los contras de `doctest` y `unittest`?"

Para los siguientes ejercicios, considera pedir ayuda a un asistente virtual, pero como siempre, recuerda probar los resultados.

18.11.2. Ejercicio

Uno de los ejercicios del Capítulo 7 pide una función llamada `uses_none` que recibe una palabra y un string de letras prohibidas, y devuelve `True` si la palabra no usa ninguna de esas letras. Aquí tienes una solución.

```
def uses_none(word, forbidden):
    for letter in word.lower():
        if letter in forbidden.lower():
            return False
    return True
```

Escribe una versión de esta función que use operaciones de `set` en lugar de un bucle `for`. Pista: pregunta a un asistente virtual: "¿Cómo calculo la intersección de conjuntos de Python?"

Puedes usar este esquema para empezar.

```
def uses_none(word, forbidden):
    """Checks whether a word avoid forbidden letters.

    >>> uses_none('banana', 'xyz')
    True
    >>> uses_none('apple', 'efg')
    False
    >>> uses_none('', 'abc')
    True
    """
    return False
```

```
from doctest import run_docstring_examples

def run_doctests(func):
    run_docstring_examples(func, globals(), name=func.__name__)
```

```
run_doctests(uses_none)
```

18.11.3. Ejercicio

Scrabble es un juego de mesa donde el objetivo es usar fichas de letras para formar palabras. Por ejemplo, si tenemos fichas con las letras `T`, `A`, `B`, `L`, `E`, podemos formar `BELT` y `LATE` usando un subconjunto de las fichas – pero no podemos formar `BEET` porque no tenemos dos `E`.

Escribe una función que reciba un string de letras y una palabra, y compruebe si las letras pueden formar la palabra, teniendo en cuenta cuántas veces aparece cada letra.

Puedes usar el siguiente esquema para empezar.

```
def can_spell(letters, word):
    """Check whether the letters can spell the word.

    >>> can_spell('table', 'belt')
    True
    >>> can_spell('table', 'late')
    True
    >>> can_spell('table', 'beet')
    False
    """
    return False
```

```
run_doctests(can_spell)
```

18.11.4. Ejercicio

En uno de los ejercicios del Capítulo 17, mi solución para `has_straightflush` usa el siguiente método, que particiona una `PokerHand` en una lista de cuatro manos, donde cada mano contiene cartas del mismo palo.

```
    def partition(self):
        """Make a list of four hands, each containing only one suit."""
        hands = []
        for i in range(4):
            hands.append(PokerHand())

        for card in self.cards:
            hands[card.suit].add_card(card)

        return hands
```

Escribe una versión simplificada de esta función usando un `defaultdict`.

Aquí tienes un esquema de la clase `PokerHand` y de la función `partition_suits` que puedes usar para empezar.

```
class PokerHand(Hand):

    def partition(self):
        return {}
```

Para probar tu código, crearemos un mazo y lo barajaremos.

```
cards = Deck.make_cards()
deck = Deck(cards)
deck.shuffle()
```

Luego crea una `PokerHand` y añádele siete cartas.

```
random_hand = PokerHand('random')

for i in range(7):
    card = deck.pop_card()
    random_hand.add_card(card)

print(random_hand)
```

Si invocas `partition` e imprimes los resultados, cada mano debería contener solo cartas de un palo.

```
hand_dict = random_hand.partition()

for hand in hand_dict.values():
    print(hand)
    print()
```

18.11.5. Ejercicio

Aquí tienes la función del Capítulo 11 que calcula números de Fibonacci.

```
def fibonacci(n):
    if n == 0:
        return 0

    if n == 1:
        return 1

    return fibonacci(n-1) + fibonacci(n-2)
```

Escribe una versión de esta función con una sola sentencia retorno que use dos expresiones condicionales, una anidada dentro de la otra.

```
fibonacci(10)    # should be 55
```

```
fibonacci(20)    # should be 6765
```

18.11.6. Ejercicio

La siguiente es una función que calcula el coeficiente binomial de forma recursiva.

```
def binomial_coeff(n, k):
    """Compute the binomial coefficient "n choose k".

    n: number of trials
    k: number of successes

    returns: int
    """
    if k == 0:
        return 1

    if n == 0:
        return 0

    return binomial_coeff(n-1, k) + binomial_coeff(n-1, k-1)
```

Reescribe el cuerpo de la función usando expresiones condicionales anidadas.

Esta función no es muy eficiente porque acaba calculando los mismos valores una y otra vez. Hazla más eficiente memoizándola, como se describe en el Capítulo 10.

```
binomial_coeff(10, 4)    # should be 210
```

18.11.7. Ejercicio

Aquí tienes el método `__str__` de la clase `Deck` en el Capítulo 17.

```
%%add_method_to Deck

    def __str__(self):
        res = []
        for card in self.cards:
            res.append(str(card))
        return '\n'.join(res)
```

Escribe una versión más concisa de este método con una comprensión de listas o una expresión generadora.

Puedes usar este ejemplo para probar tu solución.

```
cards = Deck.make_cards()
deck = Deck(cards)
print(deck)
```

Think Python: 3.ª edición

Traducción al español por midudev (Miguel Ángel Durán).

Puedes encargar versiones impresas y ebook de *Think Python 3e* en Bookshop.org y Amazon.

19. Reflexiones finales

Aprender a programar no es fácil, pero si has llegado hasta aquí, has empezado con buen pie. Ahora tengo algunas sugerencias sobre cómo puedes seguir aprendiendo y aplicar lo que has aprendido.

Este libro pretende ser una introducción general a la programación, así que no nos hemos centrado en aplicaciones específicas. Según tus intereses, hay muchas áreas en las que puedes aplicar tus nuevas habilidades.

Si te interesa la ciencia de datos, hay tres libros míos que podrían gustarte:

- *Think Stats: Exploratory Data Analysis*, O'Reilly Media, 2014.
- *Think Bayes: Bayesian Statistics in Python*, O'Reilly Media, 2021.
- *Think DSP: Digital Signal Processing in Python*, O'Reilly Media, 2016.

Si te interesan el modelado físico y los sistemas complejos, quizá te gusten:

- *Modeling and Simulation in Python: An Introduction for Scientists and Engineers*, No Starch Press, 2023.
- *Think Complexity: Complexity Science and Computational Modeling*, O'Reilly Media, 2018.

Estos libros usan NumPy, SciPy, pandas y otras librerías de Python para ciencia de datos y computación científica.

Este libro intenta encontrar un equilibrio entre los principios generales de la programación y los detalles de Python. Como resultado, no incluye todas las características del lenguaje Python. Para saber más sobre Python, y para buenos consejos sobre cómo usarlo, recomiendo *Fluent Python: Clear, Concise, and Effective Programming*, segunda edición de Luciano Ramalho, O'Reilly Media, 2022.

Después de una introducción a la programación, un siguiente paso habitual es aprender sobre estructuras de datos y algoritmos. Tengo una obra en curso sobre este tema, llamada *Data Structures and Information Retrieval in Python*. Hay una versión electrónica gratuita disponible en Green Tea Press en https://greenteapress.com.

A medida que trabajes en programas más complejos, encontrarás nuevos desafíos. Puede resultarte útil repasar las secciones de este libro sobre depuración. En particular, recuerda las seis R de la depuración del Capítulo 12 (por sus nombres en inglés): leer, ejecutar, reflexionar, explicárselo a un patito de goma, tomar distancia y descansar.

Este libro sugiere herramientas para ayudar con la depuración, incluidas las funciones `print` y `repr`, la función `structshape` del Capítulo 11 – y las funciones integradas `isinstance`, `hasattr` y `vars` del Capítulo 14.

También sugiere herramientas para probar programas, incluidas la sentencia `assert`, el módulo `doctest` y el módulo `unittest`. Incluir pruebas en tus programas es una de las mejores formas de prevenir y detectar errores, y de ahorrar tiempo en depuración.

Pero la mejor depuración es el que no tienes que hacer. Si usas un proceso de desarrollo incremental como se describe en el Capítulo 6 – y pruebas a medida que avanzas – cometerás menos errores y los encontrarás más rápido cuando aparezcan. Además, recuerda la encapsulación y la generalización del Capítulo 4, que son especialmente útiles cuando desarrollas código en Jupyter notebooks.

A lo largo de este libro, he sugerido formas de usar asistentes virtuales para ayudarte a aprender, programar y depurar. Espero que estas herramientas te estén resultando útiles.

Además de asistentes virtuales como ChatGPT, quizá también quieras usar una herramienta como Copilot, que autocompleta código mientras escribes. Al principio no recomendé usar estas

herramientas porque pueden resultar abrumadoras para principiantes. Pero quizá ahora quieras explorarlas.

Usar herramientas de AI de forma efectiva requiere algo de experimentación y reflexión para encontrar un flujo que funcione para ti. Si te resulta molesto copiar código de ChatGPT a Jupyter, quizá prefieras algo como Copilot. Pero el trabajo cognitivo que haces para componer un prompt e interpretar la respuesta puede ser tan valioso como el código que genera la herramienta, en la misma línea que la depuración con patito de goma.

A medida que ganes experiencia programando, quizá quieras explorar otros entornos de desarrollo. Creo que Jupyter notebooks es un buen lugar para empezar, pero son relativamente nuevos y no se usan tanto como los entornos de desarrollo integrados (IDE) convencionales. Para Python, los IDE más populares incluyen PyCharm y Spyder – y Thonny, que suele recomendarse para principiantes. Otros IDE, como Visual Studio Código y Eclipse, también funcionan con otros lenguajes de programación. O, como alternativa más sencilla, puedes escribir programas en Python usando cualquier editor de texto que te guste.

Mientras continúas tu viaje en la programación, no tienes que hacerlo en soledad. Si vives en una ciudad o cerca de una, es muy probable que haya un grupo de usuarios de Python al que puedas unirte. Estos grupos suelen ser amables con principiantes, así que no tengas miedo. Si no hay ningún grupo cerca de ti, quizá puedas unirte a eventos de forma remota. Además, mantente atento a conferencias regionales de Python.

Una de las mejores formas de mejorar tus habilidades de programación es aprender otro lenguaje. Si te interesan la estadística y ciencia de datos, quizá quieras aprender R. Pero recomiendo especialmente aprender un lenguaje funcional como Racket o Elixir. La programación funcional requiere una forma distinta de pensar, lo que cambia la manera en que piensas sobre los programas.

¡Buena suerte!

Think Python: 3.ª edición

Traducción al español por midudev (Miguel Ángel Durán).

Notebooks en blanco

Para cada capítulo hay un notebook en blanco con el texto del libro y la mayor parte del código eliminado. Estos notebooks son útiles para ejercicios guiados donde los estudiantes completan las celdas.

Capítulo 1: Programar como una forma de pensar

- Abrir el capítulo 1 en Colab

Capítulo 2: Variables y sentencias

- Abrir el capítulo 2 en Colab

Capítulo 3: Funciones

- Abrir el capítulo 3 en Colab

Capítulo 4: Funciones e interfaces

- Abrir el capítulo 4 en Colab

Capítulo 5: Condicionales y recursión

- Abrir el capítulo 5 en Colab

Capítulo 6: Valores de retorno

- Abrir el capítulo 6 en Colab

Capítulo 7: Iteración y búsqueda

- Abrir el capítulo 7 en Colab

Capítulo 8: Cadenas y expresiones regulares

- Abrir el capítulo 8 en Colab

Capítulo 9: Listas

- Abrir el capítulo 9 en Colab

Capítulo 10: Diccionarios

- Abrir el capítulo 10 en Colab

Capítulo 11: Tuplas

- Abrir el capítulo 11 en Colab

Capítulo 12: Análisis y generación de texto

- Abrir el capítulo 12 en Colab

Capítulo 13: Archivos y bases de datos

- Abrir el capítulo 13 en Colab

Capítulo 14: Clases y funciones

- Abrir el capítulo 14 en Colab

Capítulo 15: Clases y métodos

- Abrir el capítulo 15 en Colab

Capítulo 16: Clases y objetos

- Abrir el capítulo 16 en Colab

Capítulo 17: Herencia

- Abrir el capítulo 17 en Colab

Capítulo 18: Extras de Python

- Abrir el capítulo 18 en Colab

Capítulo 19: Reflexiones finales

- Abrir el capítulo 19 en Colab