Capítulo 1: Trabajar con números

## **1** Trabajar con números

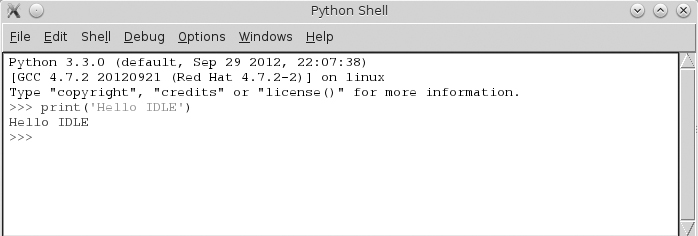


Vamos a dar nuestros primeros pasos en el uso de Python para explorar el mundo de las matemáticas y la ciencia. Ahora lo haremos de forma sencilla para que puedas familiarizarte con el uso de Python. Empezaremos realizando operaciones matemáticas básicas, y luego escribiremos programas sencillos para manipular y comprender los números. ¡Empecemos!

### **Operaciones matemáticas básicas**

El *shell interactivo de* Python va a ser nuestro amigo en este libro. Inicia el intérprete de comandos IDLE de Python 3 y di "hola" (ver [Figura 1-1](ch01.html#ch1fig1)) escribiendo print('Hello IDLE') y pulsando INTRO. (Para obtener instrucciones sobre cómo instalar Python e iniciar IDLE, consulta [el Apéndice A](app01.html#app01).) IDLE obedece tu orden e imprime las palabras en la pantalla. Enhorabuena: ¡acabas de escribir un programa!

Cuando vuelvas a ver el prompt >>>, IDLE estará listo para recibir más instrucciones.



*Figura 1-1: Shell IDLE de Python 3*

Python puede actuar como una calculadora glorificada, haciendo cálculos sencillos. Sólo tienes que escribir una expresión y Python la evaluará. Después de pulsar ENTER, el resultado aparece inmediatamente.

Pruébalo. Puedes sumar y restar números utilizando los operadores de suma (+) y resta (–). Por ejemplo:

>>> 1 + 2  
3  
>>> 1 + 3.5  
4.5  
>>> -1 + 2.5  
1.5  
>>> 100 – 45  
55  
>>> -1.1 + 5  
3.9

Para multiplicar, utiliza el operador multiplicación (\*):

>>> 3 \* 2  
6  
>>> 3.5 \* 1.5  
5.25

Para dividir, utiliza el operador división (/):

>>> 3 / 2  
1.5  
>>> 4 / 2  
2.0

Como puedes ver, cuando pides a Python que realice una operación de división, te devuelve también la parte fraccionaria del número. Si quieres el resultado en forma de número entero, sin los valores decimales, debes utilizar el operador división suelo (//):

>>> 3 // 2  
1

El operador de división piso divide el primer número entre el segundo y redondea el resultado al entero inmediatamente inferior. Esto resulta interesante cuando uno de los números es negativo. Por ejemplo:

>>> -3 // 2  
-2

El resultado final es el número entero inferior al resultado de la operación de división (-3/2 = -1.5, por lo que el resultado final es -2).

En cambio, si sólo quieres el resto, debes utilizar el operador módulo (%):

>>> 9 % 2  
1

Puedes calcular la potencia de los números utilizando el operador exponencial (\*\*). Los ejemplos siguientes lo ilustran:

>>> 2 \*\* 2  
4  
>>> 2 \*\* 10  
1024  
>>> 1 \*\* 10  
1

También podemos utilizar el símbolo exponencial para calcular potencias menores que 1. Por ejemplo, la raíz cuadrada de un número *n* puede expresarse como n1/2 y la raíz cúbica como n1/3:

>>> 8 \*\* (1/3)  
2.0

Como muestra este ejemplo, puedes utilizar paréntesis para combinar operaciones matemáticas en expresiones más complicadas. Python evaluará la expresión siguiendo la regla estándar *PEMDAS* para el orden de los cálculos: paréntesis, exponentes, multiplicación, división, suma y resta. Considera las dos expresiones siguientes: una sin paréntesis y otra con paréntesis:

>>> 5 + 5 \* 5  
30  
>>> (5 + 5) \* 5  
50

En el primer ejemplo, Python calcula primero la multiplicación: 5 por 5 es 25; 25 más 5 es 30. En el segundo ejemplo, se evalúa primero la expresión entre paréntesis, tal como cabría esperar: 5 más 5 es 10; 10 por 5 es 50.

Estos son los fundamentos absolutos de la manipulación de números en Python. Aprendamos ahora cómo podemos asignar nombres a los números.

### **Etiquetas: Asignar nombres a los números**

Cuando empecemos a diseñar programas Python más complejos, asignaremos nombres a los números, a veces por comodidad, pero sobre todo por necesidad. He aquí un ejemplo sencillo:

➊ >>> a = 3  
>>> a + 1  
4  
➋ >>> a = 5  
>>> a + 1  
6

En ➊, asignamos el nombre a al número 3. Cuando pedimos a Python que evalúe el resultado de la expresión a + 1, ve que el número al que se refiere a es el 3, y entonces suma 1 y muestra el resultado (4). En ➋, cambiamos el valor de a a 5, y esto se refleja en la segunda operación de suma. Utilizar el nombre a es conveniente porque puedes cambiar simplemente el número al que apunta a y Python utiliza este nuevo valor cuando se hace referencia a a en cualquier lugar después de eso.

Este tipo de nombre se denomina *etiqueta*. Es posible que hayas conocido el término *variable* para describir la misma idea en otro lugar. Sin embargo, teniendo en cuenta que *variable* también es un término matemático (se utiliza para referirse a algo como *x* en la ecuación *x* + 2 = 3), en este libro utilizo el término *variable* sólo en el contexto de ecuaciones y expresiones matemáticas.

### **Diferentes tipos de números**

Te habrás dado cuenta de que he utilizado dos tipos de números para demostrar las operaciones matemáticas: números sin punto decimal, que ya conoces como *números enteros*, y números con punto decimal, que los programadores llaman *números de coma flotante*. A los humanos no nos cuesta reconocer y trabajar con números, ya estén escritos como números enteros, decimales con coma flotante, fracciones o números romanos. Pero en algunos de los programas que escribimos en este libro, sólo tendrá sentido realizar una tarea con un tipo concreto de número, por lo que a menudo tendremos que escribir un poco de código para que los programas comprueben si los números que introducimos son del tipo correcto.

Python considera que los números enteros y los de coma flotante son *tipos* diferentes. Si utilizas la función type(), Python te dirá qué tipo de número acabas de introducir. Por ejemplo:

>>> type(3)  
<class 'int'>  
  
>>> type(3.5)  
<class 'float'>  
  
>>> type(3.0)  
<class 'float'>

Aquí puedes ver que Python clasifica el número 3 como un número entero (tipo 'int'), pero clasifica 3,0 como un número de coma flotante (tipo 'float'). Todos sabemos que 3 y 3,0 son matemáticamente equivalentes, pero en muchas situaciones, Python tratará estos dos números de forma diferente porque son dos tipos distintos.

Algunos de los programas que escribimos en este capítulo sólo funcionarán correctamente con un número entero como entrada. Como acabamos de ver, Python no reconocerá un número como 1,0 o 4,0 como un entero, así que si queremos aceptar números como esos como entrada válida en estos programas, tendremos que convertirlos de números de coma flotante a enteros. Por suerte, hay una función integrada en Python que hace precisamente eso:

>>> int(3.8)  
3  
>>> int(3.0)  
3

La función int() toma el número de coma flotante de entrada, elimina todo lo que venga después del punto decimal y devuelve el número entero resultante. La función float() funciona de forma similar para realizar la conversión inversa:

>>> float(3)  
3.0

float() toma el número entero de entrada y le añade un punto decimal para convertirlo en un número de coma flotante.

#### ***Trabajar con fracciones***

Python también puede manejar fracciones, pero para ello necesitaremos utilizar el módulo fractions de Python. Puedes considerar un *módulo* como un programa escrito por otra persona que puedes utilizar en tus propios programas. Un módulo puede incluir clases, funciones e incluso definiciones de etiquetas. Puede formar parte de la biblioteca estándar de Python o distribuirse desde una ubicación de terceros. En este último caso, tendrías que instalar el módulo antes de poder utilizarlo.

El módulo fractions forma parte de la biblioteca estándar, lo que significa que ya está instalado. Define una clase Fraction, que es la que utilizaremos para introducir fracciones en nuestros programas. Antes de poder utilizarla, tendremos que *importarla*, que es una forma de decirle a Python que queremos utilizar la clase de este módulo. Veamos un ejemplo rápido: crearemos una nueva etiqueta, f, que hace referencia a la fracción 3/4:

➊ >>> from fractions import Fraction  
➋ >>> f = Fraction(3, 4)  
➌ >>> f  
Fraction(3, 4)

Primero importamos la clase Fraction del módulo fractions ➊. A continuación, creamos un objeto de esta clase pasando como parámetros el numerador y el denominador ➋. Esto crea un objeto Fraction para la fracción 3/4. Cuando imprimimos el objeto ➌, Python muestra la fracción de la forma Fraction(numerator, denominator).

Las operaciones matemáticas básicas, incluidas las operaciones de comparación, son todas válidas para las fracciones. También puedes combinar una fracción, un número entero y un número de coma flotante en una misma expresión:

>>> Fraction(3, 4) + 1 + 1.5  
3.25

Cuando tienes un número de coma flotante en una expresión, el resultado de la expresión se devuelve como un número de coma flotante.

En cambio, cuando sólo tienes una fracción y un número entero en la expresión, el resultado es una fracción, aunque el resultado tenga un denominador de 1.

>>> Fraction(3, 4) + 1 + Fraction(1/4)  
Fraction(2, 1)

Ahora ya conoces lo básico para trabajar con fracciones en Python. Pasemos a otro tipo de números.

#### ***Números complejos***

Los números que hemos visto hasta ahora son los llamados *números reales*. Python también admite números *complejos* con la parte imaginaria identificada por la letra *j* o *J* (a diferencia de la letra *i* utilizada en notación matemática). Por ejemplo, el número complejo 2 + *3i* se escribiría en Python como 2 + *3j*:

>>> a = 2 + 3j  
>>> type(a)  
<class 'complex'>

Como puedes ver, cuando utilizamos la función type() sobre un número complejo, Python nos dice que se trata de un objeto del tipo complex.

También puedes definir números complejos utilizando la función complex():

>>> a = complex(2, 3)  
>>> a  
(2 + 3j)

Aquí pasamos las partes real e imaginaria del número complejo como dos argumentos a la función complex(), y ésta devuelve un número complejo.

Puedes sumar y restar números complejos del mismo modo que los números reales:

>>> b = 3 + 3j  
>>> a + b  
(5 + 6j)  
>>> a - b  
(-1 + 0j)

La multiplicación y la división de números complejos también se realizan de forma similar:

>>> a \* b  
(-3 + 15j)  
>>> a / b  
(0.8333333333333334 + 0.16666666666666666j)

Las operaciones de módulo (%) y división por el suelo (//) no son válidas para los números complejos.

Las partes real e imaginaria de un número complejo pueden obtenerse utilizando sus atributos real y imag, como se indica a continuación:

>>> z = 2 + 3j  
>>> z.real  
2.0  
>>> z.imag  
3.0

El *conjugado* de un número complejo tiene la misma parte real pero una parte imaginaria de igual magnitud y signo contrario. Se puede obtener utilizando el método conjugate():

>>> z.conjugate()  
(2 - 3j)

Tanto la parte real como la imaginaria son números en coma flotante. Utilizando las partes real e imaginaria, puedes calcular la *magnitud* de un número complejo con la siguiente fórmula, donde *x* e *y* son las partes real e imaginaria del número, respectivamente: image. En Python, sería como sigue:

>>> (z.real \*\* 2 + z.imag \*\* 2) \*\* 0.5  
3.605551275463989

Una forma más sencilla de hallar la magnitud de un número complejo es con la función abs(). La función abs() devuelve el valor absoluto cuando se invoca con un número real como argumento. Por ejemplo, tanto abs(5) como abs(-5) devuelven 5. Sin embargo, para los números complejos, devuelve la magnitud:

>>> abs(z)  
3.605551275463989

El módulo cmath de la biblioteca estándar (cmath para *matemáticas complejas*) proporciona acceso a otras funciones especializadas para trabajar con números complejos.

### **Obtener la entrada del usuario**

Cuando empecemos a escribir programas, nos será útil disponer de una forma agradable y sencilla de aceptar la entrada del usuario mediante la función input(). De esta forma, podemos escribir programas que pidan al usuario que introduzca un número, realice operaciones específicas con ese número y, a continuación, muestre los resultados de las operaciones. Veámoslo en acción:

➊ >>> a = input()  
➋ 1

En ➊, llamamos a la función input(), que espera a que escribas algo, como se muestra en ➋, y pulses INTRO. La información introducida se almacena en a:

>>> a  
➌ '1'

Observa las comillas simples alrededor del 1 en ➌. La función input() devuelve la entrada como una *cadena*. En Python, una cadena es cualquier conjunto de caracteres entre dos comillas. Cuando quieras crear una cadena, puedes utilizar comillas simples o dobles:

>>> s1 = 'a string'  
>>> s2 = "a string"

Aquí, tanto s1 como s2 se refieren a la misma cadena.

Aunque los únicos caracteres de una cadena sean números, Python no tratará esa cadena como un número a menos que nos deshagamos de esas comillas. Así que antes de poder realizar cualquier operación matemática con la entrada, tendremos que convertirla al tipo de número correcto. Una cadena puede convertirse en un número entero o de coma flotante utilizando la función int() o float(), respectivamente:

>>> a = '1'  
>>> int(a) + 1  
2  
>>> float(a) + 1  
2.0

Son las mismas funciones int() y float() que vimos antes, pero esta vez en lugar de convertir la entrada de un tipo de número a otro, toman una cadena como entrada ('1') y devuelven un número (2 o 2.0). Sin embargo, es importante tener en cuenta que la función int() no puede convertir una cadena que contenga un decimal de coma flotante en un número entero. Si tomas una cadena que contenga un número de coma flotante (como '2.5' o incluso '2.0') e introduces esa cadena en la función int(), obtendrás un mensaje de error:

>>> int('2.0')  
Traceback (most recent call last):  
  
File "<pyshell#26>", line 1, in <module>  
int('2.0')  
ValueError: invalid literal for int() with base 10: '2.0'

Éste es un ejemplo de *excepción:*la forma que tiene *Python*de decirte que no puede seguir ejecutando tu programa debido a un error. En este caso, la excepción es del tipo ValueError. (Para un rápido repaso de las excepciones, consulta [el Apéndice B](app02.html#app02).)

Del mismo modo, cuando proporcionas un número fraccionario como 3/4 como entrada, Python no puede convertirlo en un número de coma flotante o entero equivalente. Una vez más, se lanza una excepción ValueError:

>>> a = float(input())  
3/4  
Traceback (most recent call last):  
File "<pyshell#25>", line 1, in <module>  
a=float(input())  
ValueError: could not convert string to float: '3/4'

Puede resultarte útil realizar la conversión en un bloque try...except para poder *manejar* esta excepción y alertar al usuario de que el programa se ha encontrado con una entrada no válida. A continuación veremos los bloques try...except.

#### ***Manejar excepciones y entradas no válidas***

Si no estás familiarizado con try...except, la idea básica es la siguiente: si ejecutas una o más sentencias en un bloque try...except y se produce un error durante la ejecución, tu programa no se bloqueará e imprimirá un Traceback. En su lugar, la ejecución se transfiere al bloque except, donde puedes realizar una operación apropiada, por ejemplo, imprimir un mensaje de error útil o intentar otra cosa.

Así es como realizarías la conversión anterior en un bloque try...except e imprimirías un mensaje de error útil si la entrada no es válida:

>>> try:  
a = float(input('Enter a number: '))  
except ValueError:  
print('You entered an invalid number')

Ten en cuenta que tenemos que especificar el tipo de excepción que queremos manejar. Aquí, queremos manejar la excepción ValueError, así que la especificamos como except ValueError.

Ahora, cuando des una entrada no válida, como 3/4, imprime un mensaje de error útil, como se muestra en ➊:

Enter a number: 3/4  
➊ You entered an invalid number

También puedes especificar un aviso con la función input() para indicar al usuario qué tipo de entrada se espera. Por ejemplo:

>>> a = input('Input an integer: ')

El usuario verá ahora el mensaje indicándole que introduzca un número entero como entrada:

Input an integer: 1

En muchos programas de este libro, pediremos al usuario que introduzca un número como entrada, por lo que tendremos que asegurarnos de que nos ocupamos de la conversión antes de intentar realizar cualquier operación con estos números. Puedes combinar la entrada y la conversión en una sola sentencia, como se indica a continuación:

>>> a = int(input())  
1  
>>> a + 1  
2

Esto funciona muy bien si el usuario introduce un número entero. Pero como hemos visto antes, si la entrada es un número en coma flotante (incluso uno equivalente a un entero, como 1,0), se producirá un error:

>>> a = int(input())  
1.0  
Traceback (most recent call last):  
File "<pyshell#42>", line 1, in <module>  
a=int(input())  
ValueError: invalid literal for int() with base 10: '1.0'

Para evitar este error, podríamos configurar una captura ValueError como la que vimos antes para las fracciones. De este modo, el programa detectaría los números en coma flotante, lo que no funcionaría en un programa pensado para enteros. Sin embargo, también detectaría números como 1,0 y 2,0, que Python *ve* como números de coma flotante pero que son equivalentes a los enteros y funcionarían perfectamente si se introdujeran como el tipo Python correcto.

Para evitar todo esto, utilizaremos el método is\_integer() para filtrar cualquier número con un dígito significativo después del punto decimal. (Este método sólo está definido para números de tipo float en Python; no funcionará con números que ya estén introducidos en forma de entero).

He aquí un ejemplo:

>>> 1.1.is\_integer()  
False

Aquí, llamamos al método is\_integer() para comprobar si 1,1 es un número entero, y el resultado es False porque 1,1 es realmente un número en coma flotante. Por otro lado , cuando se llama al método con 1,0 como número de coma flotante, el resultado es True:

>>> 1.0.is\_integer()  
True

Podemos utilizar is\_integer() para filtrar entradas que no sean números enteros y mantener entradas como 1,0, que se expresa como un número de coma flotante pero equivale a un número entero. Veremos cómo encajaría el método en un programa más grande un poco más adelante.

#### ***Fracciones y números complejos como entrada***

La clase Fraction que hemos conocido antes también es capaz de convertir una cadena como '3/4' en un objeto Fraction. De hecho, así es como podemos aceptar una fracción como entrada:

>>> a = Fraction(input('Enter a fraction: '))  
Enter a fraction: 3/4  
>>> a  
Fraction(3, 4)

Intenta introducir una fracción como 3/0 como entrada:

>>> a = Fraction(input('Enter a fraction: '))  
Enter a fraction: 3/0  
Traceback (most recent call last):  
File "<pyshell#2>", line 1, in <module>  
a = Fraction(input('Enter a fraction: '))  
File "/usr/lib64/python3.3/fractions.py", line 167, in \_\_new\_\_  
raise ZeroDivisionError('Fraction(%s, 0)' % numerator)  
ZeroDivisionError: Fraction(3, 0)

El mensaje de excepción ZeroDivisionError te dice (como ya sabes) que una fracción con denominador 0 no es válida. Si piensas hacer que los usuarios introduzcan fracciones en uno de tus programas, es una buena idea atrapar siempre este tipo de excepciones. He aquí cómo puedes hacer algo así:

>>> try:  
a = Fraction(input('Enter a fraction: '))  
except ZeroDivisionError:  
print('Invalid fraction')  
  
Enter a fraction: 3/0  
Invalid fraction

Ahora, cada vez que el usuario de tu programa introduzca una fracción con 0 en el denominador, imprimirá el mensaje Invalid fraction.

Del mismo modo, la función complex() puede convertir una cadena como '2+3j' en un número complejo:

>>> z = complex(input('Enter a complex number: '))  
Enter a complex number: 2+3j  
>>> z  
(2+3j)

Si introduces la cadena como '2 + 3j' (con espacios), aparecerá un mensaje de error ValueError:

>>> z = complex(input('Enter a complex number: '))  
Enter a complex number: 2 + 3j  
Traceback (most recent call last):  
File "<pyshell#43>", line 1, in <module>  
z = complex(input('Enter a complex number: '))  
ValueError: complex() arg is a malformed string

Es una buena idea capturar la excepción ValueError al convertir una cadena en un número complejo, como hemos hecho para otros tipos de números.

### **Escribir programas que hagan las matemáticas por ti**

Ahora que hemos aprendido algunos de los conceptos básicos, podemos combinarlos con las sentencias condicionales y de bucle de Python para hacer algunos programas un poco más avanzados y útiles.

#### ***Calcular los factores de un número entero***

Cuando un entero distinto de cero, a, divide a otro entero, b, dejando un resto 0, se dice que a es un *factor* de b. Como ejemplo, 2 es un factor de todos los enteros pares. Podemos escribir una función como la siguiente para averiguar si un entero distinto de cero, a, es factor de otro entero, b:

>>> def is\_factor(a, b):  
if b % a == 0:  
return True  
else:  
return False

Utilizamos el operador % introducido anteriormente en este capítulo para calcular el resto. Si alguna vez te planteas una pregunta como "¿Es 4 un factor de 1024?", puedes utilizar la función is\_factor():

>>> is\_factor(4, 1024)  
True

Para cualquier número entero positivo *n*, ¿cómo encontramos todos sus factores positivos? Para cada uno de los enteros entre 1 y *n*, comprobamos el resto tras dividir *n* por ese entero. Si queda un resto de 0, es un factor. Utilizaremos la función range() para escribir un programa que recorra cada uno de esos números entre 1 y *n*.

Antes de escribir el programa completo, veamos cómo funciona range(). Un uso típico de la función range() tiene este aspecto:

>>> for i in range(1, 4):  
print(i)  
1  
2  
3

Aquí, hemos creado un bucle for y le hemos dado dos argumentos a la función rango. La función range() parte del número entero indicado como primer argumento (el *valor* de *inicio*) y continúa hasta el número entero justo *anterior* al indicado por el segundo argumento (el *valor de parada*). En este caso, le hemos dicho a Python que imprima los números de ese rango, empezando por el 1 y parando en el 4. Ten en cuenta que esto significa que Python no imprime el 4, por lo que el último número que imprime es el anterior al valor de parada (3). También es importante tener en cuenta que la función range() sólo acepta enteros como argumentos.

También puedes utilizar la función range() sin especificar el valor inicial, en cuyo caso se supone que es 0. Por ejemplo:

>>> for i in range(5):  
print(i)  
0  
1  
2  
3  
4

La diferencia entre dos enteros consecutivos producida por la función range() se conoce como valor de *paso*. Por defecto, el valor de paso es 1. Para especificar un valor de paso diferente, especifícalo como tercer argumento (el valor de inicio *no* es opcional cuando especificas un valor de paso). Por ejemplo, el siguiente programa imprime los números impares *inferiores a* 10:

>>> for i in range(1,10,2):  
print(i)  
1  
3  
5  
7  
9

Bien, ahora que hemos visto cómo funciona la función range(), estamos preparados para ver un programa de cálculo de factores. Como estoy escribiendo un programa bastante largo, en lugar de escribir este programa en el prompt interactivo de IDLE, lo escribo en el editor de IDLE. Puedes iniciar el editor seleccionando Archivo▸Nueva**ventana** en IDLE. Observa que empezamos comentando nuestro código con tres comillas simples rectas ('). El texto entre esas comillas no será ejecutado por Python como parte del programa; es sólo un comentario para nosotros, los humanos.

'''  
Find the factors of an integer  
'''  
  
def factors(b):  
  
➊ for i in range(1, b+1):  
if b % i == 0:  
print(i)  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
b = input('Your Number Please: ')  
b = float(b)  
  
➋ if b > 0 and b.is\_integer():  
factors(int(b))  
else:  
print('Please enter a positive integer')

La función factors() define un bucle for que itera una vez por cada entero entre 1 y el entero de entrada en ➊ utilizando la función range(). Aquí, queremos iterar hasta el entero introducido por el usuario, b, por lo que el valor de parada se indica como b+1. Para cada uno de estos enteros, i, el programa comprueba si divide el número introducido sin resto y lo imprime en caso afirmativo.

Cuando ejecutes este programa (seleccionando**Ejecutar▸Módulo**), te pedirá que introduzcas un número. Si tu número es un entero positivo, se imprimen sus factores. Por ejemplo:

Your Number Please: 25  
1  
5  
25

Si introduces un número no entero o un número entero negativo, el programa imprime un mensaje de error pidiéndote que introduzcas un número entero positivo:

Your Number Please: 15.5  
Please enter a positive integer

Este es un ejemplo de cómo podemos hacer que los programas sean más fáciles de usar, comprobando siempre la entrada no válida en el propio programa. Como nuestro programa sólo sirve para encontrar los factores de un entero positivo, comprobamos si el número introducido es mayor que 0 y es un entero utilizando el método is\_integer() ➋ para asegurarnos de que la entrada es válida. Si la entrada no es un entero positivo, el programa imprime una instrucción fácil de usar en lugar de un gran mensaje de error.

#### ***Generar tablas de multiplicación***

Considera tres números, *a*, *b* y *n*, donde *n* es un entero, tales que

*a* × *n* = *b*.

Podemos decir aquí que *b* es el *enésimo* *múltiplo* de *a*. Por ejemplo, 4 es el 2º múltiplo de 2, y 1024 es el 512º múltiplo de 2.

Una tabla de multiplicar de un número enumera todos los múltiplos de ese número. Por ejemplo, la tabla de multiplicar del 2 tiene este aspecto (aquí se muestran los tres primeros múltiplos):

2 × 1 = 2

2 × 2 = 4

2 × 3 = 6

Nuestro siguiente programa genera el número de la multiplicación hasta 10 para cualquier número introducido por el usuario. En este programa, utilizaremos el método format() con la función print() para que la salida del programa tenga un aspecto más agradable y legible. Por si no lo has visto antes, te explico brevemente cómo funciona.

El método format() te permite introducir etiquetas y configurarlas para que se impriman en una cadena agradable y legible, con un formato adicional alrededor. Por ejemplo, si tuviera los nombres de todas las frutas que compré en el supermercado con etiquetas separadas creadas para cada una y quisiera imprimirlas para formar una frase coherente, podría utilizar el método format() de la siguiente manera:

>>> item1 = 'apples'  
>>> item2 = 'bananas'  
>>> item3 = 'grapes'  
>>> print('At the grocery store, I bought some {0} and {1} and {2}'.format(item1, item2, item3))  
At the grocery store, I bought some apples and bananas and grapes

Primero, creamos tres etiquetas (item1, item2, y item3), cada una referida a una cadena diferente (apples, bananas, y grapes). A continuación, en la función print(), escribimos una cadena con tres marcadores de posición entre llaves: {0}, {1}, y {2}. Seguimos con .format(), que contiene las tres etiquetas que hemos creado. Esto le dice a Python que rellene esos tres marcadores de posición con los valores almacenados en esas etiquetas en el orden indicado, de modo que Python imprime el texto con {0} sustituido por la primera etiqueta, {1} sustituido por la segunda etiqueta, y así sucesivamente.

No es necesario que las etiquetas apunten a los valores que queremos imprimir. También podemos simplemente escribir los valores en .format(), como en el siguiente ejemplo:

>>> print('Number 1: {0} Number 2: {1} '.format(1, 3.578))  
Number 1: 1 Number 2: 3.578

Ten en cuenta que el número de marcadores de posición y el número de etiquetas o valores deben ser iguales.

Ahora que hemos visto cómo funciona format(), estamos preparados para echar un vistazo al programa de nuestra impresora de tablas de multiplicar:

'''  
Multiplication table printer  
'''  
  
def multi\_table(a):  
  
➊ for i in range(1, 11):  
print('{0} x {1} = {2}'.format(a, i, a\*i))  
  
if \_\_name\_\_ == '\_\_main\_\_':  
a = input('Enter a number: ')  
multi\_table(float(a))

La función multi\_table() implementa la funcionalidad principal del programa. Toma como parámetro el número para el que se imprimirá la tabla de multiplicar, a. Como queremos imprimir la tabla de multiplicar del 1 al 10, tenemos un bucle for en ➊ que itera sobre cada uno de estos números, imprimiendo el producto de sí mismo y el número, a.

Cuando ejecutas el programa, te pide que introduzcas un número, y el programa imprime su tabla de multiplicar:

Enter a number : 5  
5.0 x 1 = 5.0  
5.0 x 2 = 10.0  
5.0 x 3 = 15.0  
5.0 x 4 = 20.0  
5.0 x 5 = 25.0  
5.0 x 6 = 30.0  
5.0 x 7 = 35.0  
5.0 x 8 = 40.0  
5.0 x 9 = 45.0  
5.0 x 10 = 50.0

¿Ves qué bonita y ordenada parece esa tabla? Eso es porque hemos utilizado el método .format() para imprimir la salida según una plantilla legible y uniforme.

Puedes utilizar el método format() para controlar aún más cómo se imprimen los números. Por ejemplo, si quieres números con sólo dos decimales, puedes especificarlo con el método format(). Aquí tienes un ejemplo:

>>> '{0}'.format(1.25456)  
'1.25456'  
>>> '{0:.2f}'.format(1.25456)  
'1.25'

La primera sentencia de formato anterior simplemente imprime el número exactamente como lo hemos introducido. En la segunda sentencia, modificamos el marcador de posición a {0:.2f}, que significa que sólo queremos dos números después del punto decimal, con el f indicando un número de coma flotante. Como puedes ver, en la siguiente salida sólo hay dos números después del punto decimal. Ten en cuenta que el número se redondea si hay más números después del punto decimal de los que has especificado. Por ejemplo:

>>> '{0:.2f}'.format(1.25556)  
'1.26'

Aquí, 1,25556 se redondea a la centésima más próxima y se imprime como 1,26. Si utilizas .2f y el número que estás imprimiendo es un entero, se añaden ceros al final:

>>> '{0:.2f}'.format(1)  
'1.00'

Se añaden dos ceros porque especificamos que debíamos imprimir exactamente dos números después del punto decimal.

#### ***Conversión de unidades de medida***

El Sistema Internacional de Unidades define siete *magnitudes básicas*. Éstas se utilizan para derivar otras cantidades, denominadas *cantidades derivadas*. La longitud (incluidas la anchura, la altura y la profundidad), el tiempo, la masa y la temperatura son cuatro de las siete magnitudes básicas. Cada una de estas cantidades tiene una unidad de medida estándar: metro, segundo, kilogramo y kelvin, respectivamente.

Pero cada una de estas unidades de medida estándar tiene también múltiples unidades de medida no estándar. Estás más familiarizado con que la temperatura se indique como 30 grados Celsius o 86 grados Fahrenheit que como 303,15 kelvin. ¿Significa eso que 303,15 kelvin es tres veces más caliente que 86 grados Fahrenheit? De ninguna manera. No podemos comparar 86 grados Fahrenheit con 303,15 kelvin sólo por sus valores numéricos, porque están expresados en unidades de medida distintas, aunque midan la misma magnitud física: la temperatura. Sólo puedes comparar dos medidas de una magnitud física cuando están expresadas en la misma unidad de medida.

Las conversiones entre diferentes unidades de medida pueden ser complicadas, y por eso a menudo se te pide que resuelvas problemas que implican la conversión entre diferentes unidades de medida en el instituto. Es una buena forma de poner a prueba tus conocimientos matemáticos básicos. Pero Python también tiene muchas habilidades matemáticas y, a diferencia de algunos estudiantes de bachillerato, ¡no se cansa de calcular números una y otra vez en bucle! A continuación, exploraremos la escritura de programas que realicen esas conversiones de unidades por ti.

Empezaremos por la longitud. En Estados Unidos y el Reino Unido se suelen utilizar pulgadas y millas para medir la longitud, mientras que en la mayoría de los demás países se utilizan centímetros y kilómetros.

Una pulgada equivale a 2,54 centímetros, y puedes utilizar la operación de multiplicación para convertir una medida en pulgadas a centímetros. Luego puedes dividir la medida en centímetros por 100 para obtener la medida en metros. Por ejemplo, así es como puedes convertir 25,5 pulgadas a metros:

>>> (25.5 \* 2.54) / 100  
0.6476999999999999

Por otra parte, una milla equivale aproximadamente a 1,609 kilómetros. Así que si ves que tu destino está a 650 millas, estás a 650 × 1,609 kilómetros:

>>> 650 \* 1.609  
1045.85

Ahora echemos un vistazo a la conversión de *la* temperatura: convertir la temperatura de Fahrenheit a Celsius y viceversa. La temperatura expresada en Fahrenheit se convierte en su valor equivalente en Celsius mediante la fórmula

image

*F* es la temperatura en Fahrenheit, y *C* es su equivalente en Celsius. Sabes que se dice que 98,6 grados Fahrenheit es la temperatura normal del cuerpo humano. Para hallar la temperatura correspondiente en grados Celsius, evaluamos la fórmula anterior en Python:

>>> F = 98.6  
>>> (F - 32) \* (5 / 9)  
37.0

Primero, creamos una etiqueta, F, con la temperatura en Fahrenheit, 98,6. Después, evaluamos la fórmula para convertir esta temperatura a su equivalente en Celsius, que resulta ser 37,0 grados Celsius.

Para convertir la temperatura de Celsius a Fahrenheit, utilizarías la fórmula

image

Puedes evaluar esta fórmula de forma similar:

>>> C = 37  
>>> C \* (9 / 5) + 32  
98.60000000000001

Creamos una etiqueta, C, con el valor 37 (la temperatura normal del cuerpo humano en Celsius). Luego, lo convertimos en Fahrenheit utilizando la fórmula, y el resultado es 98,6 grados.

Es una faena tener que escribir estas fórmulas de conversión una y otra vez. Escribamos un programa de conversión de unidades que haga las conversiones por nosotros. Este programa presentará un menú para que los usuarios puedan seleccionar la conversión que desean realizar, pedirá los datos pertinentes y, a continuación, imprimirá el resultado calculado. El programa se muestra a continuación:

'''  
Unit converter: Miles and Kilometers  
'''  
  
def print\_menu():  
print('1. Kilometers to Miles')  
print('2. Miles to Kilometers')  
  
def km\_miles():  
km = float(input('Enter distance in kilometers: '))  
miles = km / 1.609  
  
print('Distance in miles: {0}'.format(miles))  
  
def miles\_km():  
miles = float(input('Enter distance in miles: '))  
km = miles \* 1.609  
  
print('Distance in kilometers: {0}'.format(km))  
  
if \_\_name\_\_ == '\_\_main\_\_':  
➊ print\_menu()  
➋ choice = input('Which conversion would you like to do?: ')  
if choice == '1':  
km\_miles()  
  
if choice == '2':  
miles\_km()

Este programa es un poco más largo que los otros, pero no te preocupes. En realidad es sencillo. Empecemos por ➊. Se llama a la función print\_menu(), que imprime un menú con dos opciones de conversión de unidades. En ➋, se pide al usuario que seleccione una de las dos conversiones. Si la elección es 1 (kilómetros a millas), se llama a la función km\_miles(). Si la elección es 2 (millas a kilómetros), se llama a la función miles\_km(). En ambas funciones, primero se pide al usuario que introduzca una distancia en la unidad elegida para la conversión (kilómetros para km\_miles() y millas para miles\_km()). A continuación, el programa realiza la conversión utilizando la fórmula correspondiente y muestra el resultado.

Aquí tienes un ejemplo de ejecución del programa:

1. Kilometers to Miles  
2. Miles to Kilometers  
➊ Which conversion would you like to do?: 2  
Enter distance in miles: 100  
Distance in kilometers: 160.900000

Se pide al usuario que introduzca una elección en ➊. La elección se introduce como 2 (millas a kilómetros). A continuación, el programa pide al usuario que introduzca la distancia en millas a convertir en kilómetros e imprime la conversión.

Este programa sólo convierte entre millas y kilómetros, pero en un reto de programación posterior, ampliarás este programa para que pueda realizar conversiones de otras unidades.

#### ***Encontrar las raíces de una ecuación cuadrática***

¿Qué haces cuando tienes una ecuación como *x* + 500 - 79 = 10 y necesitas encontrar el valor de la variable desconocida, *x*? Reorganizas los términos de forma que sólo tengas las constantes (500, -79 y 10) en un lado de la ecuación y la variable*(x*) en el otro lado. El resultado es la siguiente ecuación: *x* = 10 - 500 + 79.

Encontrar el valor de la expresión de la derecha te da el valor de *x*, tu solución, que también se llama *raíz* de esta ecuación. En Python, puedes hacerlo de la siguiente manera:

>>> x = 10 - 500 + 79  
>>> x  
-411

Este es un ejemplo de *ecuación lineal*. Una vez que has reordenado los términos de ambos lados, la expresión es bastante sencilla de evaluar. En cambio, para ecuaciones como x2 +*2x* + 1 = 0, encontrar las raíces de *x* suele implicar evaluar una expresión compleja conocida como *fórmula cuadrática*. Tales ecuaciones se conocen como *ecuaciones* cuadráticas, generalmente expresadas como ax2 + *bx* + *c* = 0, donde *a*, *b* y *c* son constantes. La fórmula cuadrática para calcular las raíces es la siguiente:

image

Una ecuación cuadrática tiene dos raíces -dos valores de *x* para los que los dos lados de la ecuación cuadrática son iguales (aunque a veces estos dos valores pueden resultar ser el mismo). Esto se indica aquí mediante x1 y x2 en la fórmula cuadrática.

Comparando la ecuación x2 +*2x* + 1 = 0 con la ecuación cuadrática genérica, vemos que *a* = 1, *b* = 2 y *c* = 1. Podemos sustituir estos valores directamente en la fórmula cuadrática para calcular el valor de x1 y x2. En Python, primero almacenamos los valores de *a*, *b* y *c* como las etiquetas a, b y c con los valores adecuados:

>>> a = 1  
>>> b = 2  
>>> c = 1

Luego, considerando que ambas fórmulas tienen el término b2 - *4ac*, definiremos una nueva etiqueta con *D*, tal que image:

>>> D = (b\*\*2 – 4\*a\*c)\*\*0.5

Como puedes ver, evaluamos la raíz cuadrada de b2 - *4ac* elevándola a la 0,5ª potencia. Ahora podemos escribir las expresiones para evaluar x1 y x2:

>>> x\_1 = (-b + D)/(2\*a)  
>>> x\_1  
-1.0  
>>> x\_2 = (-b - D)/(2\*a)  
>>> x\_2  
-1.0

En este caso, los valores de ambas raíces son iguales, y si sustituyes ese valor en la ecuación x2 +*2x* + 1, la ecuación se evaluará a 0.

Nuestro siguiente programa combina todos estos pasos en una función roots(), que toma los valores de *a*, *b* y *c* como parámetros, calcula las raíces y las imprime:

'''  
Quadratic equation root calculator  
'''  
  
def roots(a, b, c):  
  
D = (b\*b - 4\*a\*c)\*\*0.5  
x\_1 = (-b + D)/(2\*a)  
x\_2 = (-b - D)/(2\*a)  
  
print('x1: {0}'.format(x\_1))  
print('x2: {0}'.format(x\_2))  
  
if \_\_name\_\_ == '\_\_main\_\_':  
a = input('Enter a: ')  
b = input('Enter b: ')  
c = input('Enter c: ')  
roots(float(a), float(b), float(c))

Al principio, utilizamos las etiquetas a, b, y c para referenciar los valores de las tres constantes de una ecuación cuadrática. A continuación, llamamos a la función roots() con estos tres valores como argumentos (después de convertirlos a números en coma flotante). Esta función introduce a, b y c en la fórmula cuadrática, encuentra las raíces de esa ecuación y las imprime.

Cuando ejecutes el programa, pedirá al usuario que introduzca los valores de *a*, *b* y *c* correspondientes a una ecuación cuadrática de la que quiera hallar las raíces.

Enter a: 1  
Enter b: 2  
Enter c: 1  
  
x1: -1.000000  
x2: -1.000000

Intenta resolver algunas ecuaciones cuadráticas más con diferentes valores para las constantes, y el programa encontrará las raíces correctamente.

Lo más probable es que sepas que las ecuaciones cuadráticas también pueden tener números complejos como raíces. Por ejemplo, las raíces de la ecuación x2 + *x* + 1 = 0 son ambas números complejos. El programa anterior también puede encontrarlas. Intentémoslo ejecutando de nuevo el programa (las constantes son *a* = 1, *b* = 1 y *c* = 1):

Enter a: 1  
Enter b: 1  
Enter c: 1  
x1: (-0.49999999999999994+0.8660254037844386j)  
x2: (-0.5-0.8660254037844386j)

Las raíces impresas arriba son números complejos (indicados con j), y el programa no tiene ningún problema para calcularlas o mostrarlas.

### **Lo que has aprendido**

¡Buen trabajo al terminar el primer capítulo! Aprendiste a escribir programas que reconocen números enteros, números de coma flotante, números fraccionarios (expresados como fracción o como número de coma flotante) y números complejos. Escribiste programas que generan tablas de multiplicar, realizan conversiones de unidades y encuentran las raíces de una ecuación cuadrática. Seguro que ya estás entusiasmado por haber dado los primeros pasos para escribir programas que hagan cálculos matemáticos por ti. Antes de seguir adelante, aquí tienes algunos retos de programación que te darán la oportunidad de seguir aplicando lo que has aprendido.

### **Retos de programación**

Aquí tienes algunos retos que te darán la oportunidad de practicar los conceptos de este capítulo. Cada problema puede resolverse de múltiples maneras, pero puedes encontrar ejemplos de soluciones en [*http://www.nostarch.com/doingmathwithpython/.*](http://www.nostarch.com/doingmathwithpython/)

#### ***#nº 1: Máquina expendedora par-impar***

Intenta escribir una "máquina expendedora par-impar", que tomará un número como entrada y hará dos cosas:

1. 1. Imprimir si el número es par o impar.

2. 2. Mostrar el número seguido de los 9 números pares o impares siguientes.

Si la entrada es 2, el programa debe imprimir even y después 2, 4, 6, 8, 10, 12, 14, 16, 18, 20. Del mismo modo, si la entrada es 1, el programa debe imprimir odd y después 1, 3, 5, 7, 9, 11, 13, 15, 17, 19.

Tu programa debe utilizar el método is\_integer() para mostrar un mensaje de error si la entrada es un número con cifras significativas más allá del punto decimal.

#### ***#2: Generador de tablas de multiplicar mejorado***

Nuestro generador de tablas de multiplicar es genial, pero sólo imprime los 10 primeros múltiplos. Mejora el generador para que el usuario pueda especificar tanto el número como hasta *qué* múltiplo. Por ejemplo, debería poder introducir que quiero ver una tabla con los 15 primeros múltiplos de 9.

#### ***#3: Conversor de unidades mejorado***

El programa de conversión de unidades que escribimos en este capítulo se limita a conversiones entre kilómetros y millas. Prueba a ampliar el programa para convertir entre unidades de masa (como kilogramos y libras) y entre unidades de temperatura (como Celsius y Fahrenheit).

#### ***#4: Calculadora de fracciones***

Escribe una calculadora que pueda realizar las operaciones matemáticas básicas con dos fracciones. Debe pedir al usuario dos fracciones y la operación que quiere realizar. Como adelanto, aquí tienes cómo puedes escribir el programa sólo con la operación de suma:

'''  
Fraction operations  
'''  
from fractions import Fraction  
  
def add(a, b):  
print('Result of Addition: {0}'.format(a+b))  
  
if \_\_name\_\_ == '\_\_main\_\_':  
➊ a = Fraction(input('Enter first fraction: '))  
➋ b = Fraction(input('Enter second fraction: '))  
op = input('Operation to perform - Add, Subtract, Divide, Multiply: ')  
if op == 'Add':  
add(a,b)

Ya has visto la mayoría de los elementos de este programa. En ➊ y ➋, pedimos al usuario que introduzca las dos fracciones. A continuación, preguntamos al usuario qué operación debe realizar con las dos fracciones. Si el usuario introduce 'Add' como entrada, llamamos a la función add(), que hemos definido para hallar la suma de las dos fracciones pasadas como argumentos. La función add() realiza la operación e imprime el resultado. Por ejemplo:

Enter first fraction: 3/4  
Enter second fraction: 1/4  
Operation to perform - Add, Subtract, Divide, Multiply: Add  
Result of Addition: 1

Prueba a añadir soporte para otras operaciones como la resta, la división y la multiplicación. Por ejemplo, así es como tu programa debería poder calcular la diferencia de dos fracciones:

Enter first fraction: 3/4  
Enter second fraction: 1/4  
Operation to perform - Add, Subtract, Divide, Multiply: Subtract  
Result of Subtraction: 2/4

En el caso de la división, debes permitir que el usuario sepa si la primera fracción está dividida por la segunda fracción o viceversa.

#### ***#nº 5: Dar poder de salida al usuario***

Todos los programas que hemos escrito hasta ahora funcionan sólo para una iteración de entrada y salida. Por ejemplo, considera el programa para imprimir la tabla de multiplicar: el usuario ejecuta el programa e introduce un número; entonces el programa imprime la tabla de multiplicar y sale. Si el usuario quisiera imprimir la tabla de multiplicar de otro número, tendría que volver a ejecutar el programa.

Sería más cómodo que el usuario pudiera elegir entre salir o seguir utilizando el programa. La clave para escribir este tipo de programas es configurar un *bucle infinito*, o un bucle que no salga a menos que se le pida explícitamente que lo haga. A continuación, puedes ver un ejemplo de la disposición de un programa de este tipo:

'''  
Run until exit layout  
'''  
  
def fun():  
print('I am in an endless loop')  
  
if \_\_name\_\_ == '\_\_main\_\_':  
➊ while True:  
fun()  
➋ answer = input('Do you want to exit? (y) for yes ')  
if answer == 'y':  
break

Definimos un bucle infinito utilizando while True en ➊. Un bucle while continúa ejecutándose a menos que la condición se evalúe como False. Como hemos elegido que la condición del bucle sea el valor constante True, seguirá ejecutándose eternamente a menos que lo interrumpamos de algún modo. Dentro del bucle, llamamos a la función fun(), que imprime la cadena I am in an endless loop. En ➋, se pregunta al usuario "¿Quieres salir?". Si el usuario introduce y como entrada, el programa sale del bucle utilizando la sentencia break (break sale del bucle más interno sin ejecutar ninguna otra sentencia en ese bucle). Si el usuario introduce cualquier otra entrada (o ninguna, sólo pulsa ENTER), el bucle while continúa la ejecución de , es decir, imprime de nuevo la cadena y continúa haciéndolo hasta que el usuario desee salir. Aquí tienes un ejemplo de ejecución del programa:

I am in an endless loop  
Do you want to exit? (y) for yes n  
I am in an endless loop  
Do you want to exit? (y) for yes n  
I am in an endless loop  
Do you want to exit? (y) for yes n  
I am in an endless loop  
Do you want to exit? (y) for yes y

Basándonos en este ejemplo, vamos a reescribir el generador de tablas de multiplicar para que siga ejecutándose hasta que el usuario desee salir. La nueva versión del programa se muestra a continuación:

'''  
Multiplication table printer with  
exit power to the user  
'''  
  
def multi\_table(a):  
  
for i in range(1, 11):  
print('{0} x {1} = {2}'.format(a, i, a\*i))  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
  
while True:  
a = input('Enter a number: ')  
multi\_table(float(a))  
  
answer = input('Do you want to exit? (y) for yes ')  
if answer == 'y':  
break

Si comparas este programa con el que escribimos anteriormente, verás que el único cambio es la adición del bucle while, que incluye el indicador que pide al usuario que introduzca un número y la llamada a la función multi\_table().

Cuando ejecutes el programa, te pedirá un número e imprimirá su tabla de multiplicar, como antes. Sin embargo, posteriormente también preguntará si el usuario desea salir del programa. Si el usuario no desea salir, el programa estará listo para imprimir la tabla para otro número. Aquí tienes un ejemplo de ejecución:

Enter a number: 2  
2.000000 x 1.000000 = 2.000000  
2.000000 x 2.000000 = 4.000000  
2.000000 x 3.000000 = 6.000000  
2.000000 x 4.000000 = 8.000000  
  
2.000000 x 5.000000 = 10.000000  
2.000000 x 6.000000 = 12.000000  
2.000000 x 7.000000 = 14.000000  
2.000000 x 8.000000 = 16.000000  
2.000000 x 9.000000 = 18.000000  
2.000000 x 10.000000 = 20.000000  
Do you want to exit? (y) for yes n  
Enter a number:

Intenta reescribir algunos de los otros programas de este capítulo para que continúen ejecutándose hasta que el usuario les pida que salgan.