2.2. Tipos de datos

Vamos a efectuar un experimento de resultado curioso:

```
>>> 3 / 2↲
1
```

¡El resultado de dividir 3 entre 2 no debería ser 1, sino 1.5![3] ¿Qué ha pasado? ¿Se ha equivocado Python? No. Python ha actuado siguiendo unas reglas precisas en las que participa un nuevo concepto: el de *tipo de dato*.

2.2.1. Enteros y flotantes

Cada valor utilizado por Python es de un tipo determinado. Hasta el momento sólo hemos utilizado datos de *tipo entero*, es decir, sin decimales. Cuando se efectúa una operación, Python tiene en cuenta el tipo de los operandos a la hora de producir el resultado. *Si los dos operandos son de tipo entero, el resultado también es de tipo entero*, así que la división entera entre los enteros 3 y 2 produce el valor entero 1.

Si deseamos obtener resultados de *tipo real*, deberemos usar operandos reales. Los operandos reales deben llevar, en principio, una parte decimal, *aunque ésta sea nula*.

```
>>> 3.0 / 2.0↲
1.5
```

Hay diferencias entre enteros y reales en Python más allá de que los primeros no tengan decimales y los segundos sí. El número 3 y el número 3.0, por ejemplo, son indistinguibles en matemáticas, pero sí son diferentes en Python. ¿Qué diferencias hay?

- Los enteros suelen ocupar menos memoria.
- Las operaciones entre enteros son, generalmente, más rápidas.

Así pues, utilizaremos enteros a menos que de verdad necesitemos números con decimales.

Hemos de precisar algo respecto a la denominación de los números con decimales: el término «reales» no es adecuado, ya que induce a pensar en los números reales de las matemáticas. En matemáticas, los números reales pueden presentar infinitos decimales, y eso es imposible en un computador. Al trabajar con computadores tendremos que conformarnos con meras *aproximaciones* a los números reales.

Recuerda que todo en el computador son secuencias de ceros y unos. Deberemos, pues, representar internamente con ellos las aproximaciones a los números reales. Para facilitar el intercambio de datos, todos los computadores convencionales utilizan una misma codificación, es decir, representan del mismo modo las aproximaciones a los números reales. Esta codificación se conoce como «IEEE Standard 754 floating point» (que se puede traducir por «Estándar IEEE 754 para coma flotante»), así que llamaremos *números en formato de coma flotante* o simplemente *flotantes* a los números con decimales que podemos representar con el ordenador.

Un número flotante debe especificarse siguiendo ciertas reglas. En principio, consta de dos partes: *mantisa* y *exponente*. El exponente se separa de la *mantisa* con la letra «e» (o «E»). Por ejemplo, el número flotante 2e3 (o 2E3) tiene mantisa 2 y exponente 3, y representa al número $2 \cdot 10^3$, es decir, 2000.

[3] Una advertencia sobre convenios tipográficos. En español, la parte fraccionaria de un número se separa de la parte entera por una coma, y no por un punto. Sin embargo, la norma anglosajona indica que debe utilizarse el punto. Python sigue esta norma, así que el número que en español se denota como 1,5 debe escribirse como 1.5 para que Python lo interprete correctamente. En aras de evitar confusiones, utilizaremos siempre el punto como carácter de separación entre parte entera y fraccionaria de un número.

El exponente puede ser negativo: 3.2e-3 es $3.2 \cdot 10^{-3}$, o sea, 0.0032. Ten en cuenta que si un número flotante no lleva exponente *debe* llevar parte fraccionaria. ¡Ah! Un par de reglas más: si la parte entera del número es nula, el flotante puede empezar directamente con un punto, y si la parte fraccionaria es nula, puede acabar con un punto. Veamos un par de ejemplos: el número 0.1 se puede escribir también como .1; por otra parte, el número 2.0 puede escribirse como 2., es decir, en ambos casos el cero es opcional. ¿Demasiadas reglas? No te preocupes, con la práctica acabarás recordándolas.

IEEE Standard 754

Un número en coma flotante presenta tres componentes: el signo, la mantisa y el exponente. He aquí un número en coma flotante: -14.1×10^{-3}. El signo es negativo, la mantisa es 14.1 y el exponente es −3. Los números en coma flotante *normalizada* presentan una mantisa menor o igual que 10. El mismo número de antes, en coma flotante normalizada, es $-1.41 \times 10-2$. Una notación habitual para los números en coma flotante sustituye el producto (×) y la base del exponente por la letra «e» o «E». Notaríamos con -1.41e-2 el número del ejemplo.

Los flotantes de Python siguen la norma IEEE Standard 754. Es una codificación binaria y normalizada de los números en coma flotante y, por tanto, con base 2 para el exponente y mantisa de valor menor que 2. Usa 32 bits (precisión simple) o 64 bits (precisión doble) para codificar cada número. Python utiliza el formato de doble precisión. En el formato de precisión doble se reserva 1 bit para el signo del número, 11 para el exponente y 52 para la mantisa. Con este formato pueden representarse números tan próximos a cero como 10^{-323} (322 ceros tras el punto decimal y un uno) o de valor absoluto tan grande como 10^{308}.

No todos los números tienen una representación exacta en el formato de coma flotante. Observa qué ocurre en este caso:

```
>>> 0.1 ↲
0.10000000000000001
```

La mantisa, que vale 1/10, no puede representarse exactamente. En binario obtenemos la secuencia periódica de bits

0.0001100110011001100110011001100110011001100110011...

No hay, pues, forma de representar 1/10 con los 52 bits del formato de doble precisión. En base 10, los 52 primeros bits de la secuencia nos proporcionan el valor

0.1000000000000000055511151231257827021181583404541015625.

Es lo más cerca de 1/10 que podemos estar. En pantalla, Python sólo nos muestra sus primeros 17 decimales (con el redondeo correspondiente).

Una peculiaridad adicional de los números codificados con la norma IEEE 754 es que su precisión es diferente según el número representado: cuanto más próximo a cero, mayor es la precisión. Para números muy grandes se pierde tanta precisión que no hay decimales (¡ni unidades, ni decenas...!). Por ejemplo, el resultado de la suma 100000000.0+0.000000001 es 100000000.0, y no 100000000.000000001, como cabría esperar.

A modo de conclusión, has de saber que al trabajar con números flotantes es posible que se produzcan pequeños errores en la representación de los valores y durante los cálculos. Probablemente esto te sorprenda, pues es *vox populi* que «los ordenadores nunca se equivocan».

Es posible mezclar en una misma expresión datos de tipos distintos.

```
>>> 3.0 / 2 ↲
1.5
```

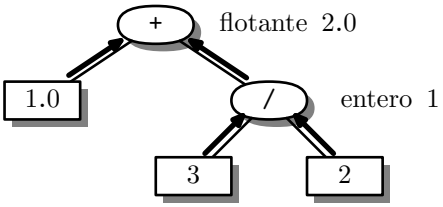
Python sigue una regla sencilla: si hay datos de tipos distintos, el resultado es del tipo «más general». Los flotantes son de tipo «más general» que los enteros.

```
>>> 1 + 2 + 3 + 4 + 5 + 6 + 0.5↲
21.5
>>> 1 + 2 + 3 + 4 + 5 + 6 + 0.0↲
21.0
```

Pero, ¡atención!, puede parecer que la regla no se observa en este ejemplo:

```
>>> 1.0 + 3 / 2↲
2.0
```

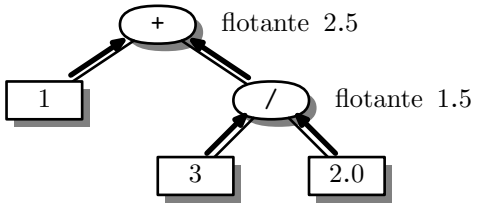
El resultado debiera haber sido 2.5, y no 2.0. ¿Qué ha pasado? Python evalúa la expresión paso a paso. Analicemos el árbol sintáctico de esa expresión:



La división es prioritaria frente a la suma, por lo que ésta se lleva a cabo en primer lugar. La división tiene dos operandos, ambos *de tipo entero*, así que produce un resultado de tipo entero: el valor 1. La suma recibe, pues, un operando flotante (el de su izquierda) de valor 1.0, y otro entero (el que resulta de la división), de valor 1. El resultado es un flotante y su valor es 2.0. ¿Qué pasaría si ejecutáramos 1 + 3 / 2.0?

```
>>> 1 + 3 / 2.0↲
2.5
```

El árbol sintáctico es, en este caso,



Así pues, la división proporciona un resultado flotante, 1.5, que al ser sumado al entero 1 de la izquierda proporciona un nuevo flotante: 2.5.

EJERCICIOS

▶ 16 ¿Qué resultará de evaluar las siguientes expresiones? Presta especial atención al tipo de datos que resulta de cada operación individual. Haz los cálculos a mano ayudándote con árboles sintácticos y comprueba el resultado con el ordenador.

a) 1 / 2 / 4.0

b) 1 / 2.0 / 4.0

c) 1 / 2.0 / 4

d) 1.0 / 2 / 4

e) 4 ** .5

f) 4.0 ** (1 / 2)

g) 4.0 ** (1 / 2) + 1 / 2

h) 4.0 ** (1.0 / 2) + 1 / 2.0

i) 3e3 / 10

j) 10 / 5e-3

k) 10 / 5e-3 + 1 l) 3 / 2 + 1

2.2.2. Valores lógicos

Desde la versión 2.3, Python ofrece un tipo de datos especial que permite expresar sólo dos valores: cierto y falso. El valor cierto se expresa con *True* y el valor falso con *False*. Son los *valores lógicos* o *booleanos*. Este último nombre deriva del nombre de un matemático, Boole, que desarrolló un sistema algebraico basado en estos dos valores y tres operaciones: la conjunción, la disyunción y la negación. Python ofrece soporte para estas operaciones con los *operadores lógicos*.