Capítulo 2

Una calculadora avanzada

—¿Sabes sumar? —le preguntó la Reina Blanca.— ¿Cuánto es uno más uno más uno más uno más uno más uno más uno más uno más uno más uno?
—No lo sé —dijo Alicia—. Perdí la cuenta.
—No sabe hacer una adición —le interrumpió la Reina Roja.

Alicia en el país de las maravillas, Lewis Carroll

El objetivo de este capítulo es que te familiarices con el *entorno interactivo* de Python, que aprendas a construir *expresiones aritméticas* almacenando los resultados en *variables* mediante *asignaciones* y que conozcas los *tipos de datos* básicos del lenguaje de programación Python.

2.1. Sesiones interactivas

Cuando programamos utilizamos un conjunto de herramientas al que denominamos *entorno de programación*. Entre estas herramientas tenemos editores de texto (que nos permiten escribir programas), compiladores o intérpretes (que traducen los programas a código de máquina), depuradores (que ayudan a detectar errores), analizadores de tiempo de ejecución (para estudiar la eficiencia de los programas), herramientas para la ejecución de pruebas unitarias (para asegurarnos de que no introducimos nuevos errores al ir desarrollando), analizadores de cobertura (para asegurarnos de que todo el código ha sido puesto a prueba), generadores de documentación (que generan documentación en formatos como HTML a partir de comentarios en los programas), analizadores estáticos (que detectan patrones típicos en código erróneo y nos advierten de problemas potenciales), sistemas de control de versiones (que permiten que varios programadores colaboren en un mismo programa, recuperar cualquier versión del programa, crear ramas de desarrollo en paralelo...), etc.

Los lenguajes interpretados suelen ofrecer una herramienta de ejecución *interactiva*. Con ella es posible dar órdenes directamente al intérprete y obtener una respuesta inmediata para cada una de ellas. Es decir, no es necesario escribir un programa completo para empezar a obtener resultados de ejecución, sino que podemos «dialogar» con el intérprete de nuestro lenguaje de programación: le pedimos que ejecute una orden y nos responde con su resultado. El entorno interactivo es de gran ayuda para experimentar con fragmentos de programa antes de incluirlos en una versión definitiva.

En esta sección veremos cómo realizar sesiones de trabajo interactivo con Python[1]. Arrancaremos el intérprete interactivo de modo distinto según el sistema operativo con el que estemos trabajando:

- Si estamos trabajando en un sistema Unix (como cualquiera de las variantes de Linux o Mac OS X), tendremos que iniciar primero un intérprete de órdenes en un terminal. Busca en los menús o barra de aplicaciones algún icono denominado «terminal» o «bash». En la ventana que se abrirá al iniciar el terminal, escribe `python` y pulsa retorno de carro. Si

[1] Abusando del lenguaje, llamaremos indistintamente Python al *entorno de programación*, al *intérprete del lenguaje* y al propio *lenguaje de programación*.

Introducción a la programación con Python 3 - UJI - DOI: http://dx.doi.org/10.6035/Sapientia93

la versión de Python que se ejecuta no es la 3.1 (o superior), sal del intérprete escribiendo `quit()` y pulsando el retorno de carro para, a continuación, escribir `python3` y pulsar nuevamente retorno de carro.

- En Microsoft Windows puedes hacer una de dos cosas:
  - ir al menú de aplicaciones y seleccionar el icono `Python (command line)` en la carpeta `Python 3.x` del menú `Todos los programas`;
  - pulsar las tecla Windows y R simultáneamente para que aparezca un cuadro de diálogo con título `Ejecutar`; escribir en la caja de texto `cmd` y pulsar el botón `Aceptar`; en la ventana que aparece escribe entonces `python` y pulsa el retorno de carro.

El sistema nos responderá dando un mensaje informativo sobre la versión de Python que estamos utilizando (y cuándo fue compilada, con qué compilador, etc.) y, a continuación, mostrará el *prompt*. El resultado será parecido a esto:

```
Python 3.2.3 (default, Feb 27 2014, 21:31:18)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> ↵
```

El *prompt* es la serie de caracteres «>>>» que aparece en la última línea. El *prompt* indica que el intérprete de Python espera que nosotros introduzcamos una orden utilizando el teclado.

Escribamos una expresión aritmética, por ejemplo 2+2, y pulsemos la tecla de retorno de carro. Cuando mostremos sesiones interactivas destacaremos el texto que teclea el usuario con texto de color azul y representaremos con el símbolo ↵ la pulsación de la tecla de retorno de carro. Python *evalúa* la expresión (es decir, obtiene su resultado) y responde mostrando el resultado por pantalla.

```
Python 3.2.3 (default, Feb 27 2014, 21:31:18)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2↵
4
>>> ↵
```

La última línea es, nuevamente, el *prompt*: Python acabó de ejecutar la última orden (evaluar una expresión y mostrar el resultado) y nos pide que introduzcamos una nueva orden.

Si deseamos acabar la sesión interactiva y salir del intérprete Python, debemos introducir una marca de final de fichero, que en Unix se indica pulsando la tecla de control y, sin soltarla, también la tecla `d`. (De ahora en adelante representaremos una combinación de teclas como la descrita así: `C-d`).

Final de fichero

La «marca de final de fichero» indica que un fichero ha terminado. ¡Pero nosotros no trabajamos con un fichero, sino con el teclado! En realidad, el ordenador considera al teclado como un fichero. Cuando deseamos «cerrar el teclado» para una aplicación, enviamos una marca de final de fichero desde el teclado. En Unix, la marca de final de fichero se envía pulsando la tecla de control y la tecla d simultáneamente, lo que indicamos con `C-d`; en Microsoft Windows, el final de fichero se indica con `C-z`.

Existe otro modo de finalizar la sesión; escribe `quit()` en el intérprete y la sesión se cerrará. En inglés, «quit» significa «abandonar».

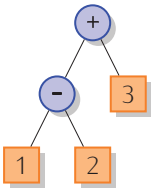
2.1.1. Los operadores aritméticos

Las *operaciones* de suma y resta, por ejemplo, se denotan con los símbolos u *operadores* `+` y `-`, respectivamente, y operan sobre dos valores numéricos (los *operandos*). Probemos algunas expresiones formadas con estos dos operadores:

```
>>> 1 + 2 ↵
3
>>> 1 + 2 + 3 ↵
6
>>> 1 - 2 + 3 ↵
2
```

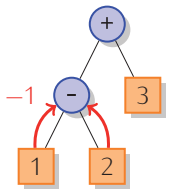
Observa que puedes introducir varias operaciones en una misma línea o expresión. El orden en que se efectúan las operaciones es (en principio) de izquierda a derecha. Por ejemplo, la expresión `1 - 2 + 3` equivale matemáticamente a $((1 - 2) + 3)$; por ello decimos que la suma y la resta son operadores *asociativos por la izquierda*.

Podemos representar gráficamente el orden de aplicación de las operaciones utilizando *árboles sintácticos*. Un árbol sintáctico es una representación gráfica en la que disponemos los operadores y los operandos como nodos y en los que cada operador está conectado a sus operandos. El árbol sintáctico de la expresión «`1 - 2 + 3`» es este:

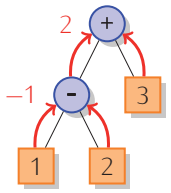


El nodo superior de un árbol recibe el nombre de *nodo raíz*. Los nodos etiquetados con operadores (representados con círculos) se denominan *nodos interiores*. Los nodos interiores tienen uno o más *nodos hijo* o *descendientes* (de los que ellos son sus respectivos *nodos padre* o *ascendientes*). Dichos nodos son nodos raíz de otros (sub)árboles sintácticos (¡la definición de árbol sintáctico es auto-referencial!). Los valores resultantes de evaluar las expresiones asociadas a dichos (sub)árboles constituyen los operandos de la operación que representa el nodo interior. Los nodos sin descendientes se denominan *nodos terminales* u *hojas* (representados con cuadrados) y corresponden a valores numéricos.

La evaluación de cada operación individual en el árbol sintáctico «fluye» de las hojas hacia la raíz (el nodo superior); es decir, en primer lugar se evalúa la subexpresión «`1 - 2`», que corresponde al subárbol más profundo. El resultado de la evaluación es −1:



A continuación se evalúa la subexpresión que suma el resultado de evaluar «`1 - 2`» al valor 3:

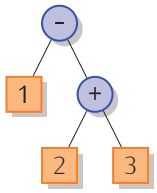


Así se obtiene el resultado final: el valor 2.

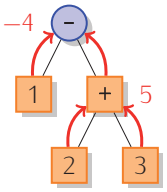
Si deseamos calcular $(1 - (2 + 3))$ podemos hacerlo añadiendo paréntesis a la expresión aritmética:

```
>>> 1 - (2 + 3)↵
-4
```

El árbol sintáctico de esta nueva expresión es

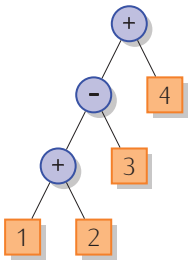


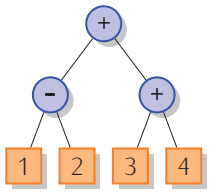
En este nuevo árbol, la primera subexpresión evaluada es la que corresponde al subárbol derecho:

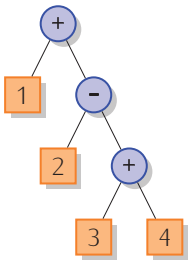


Observa que en el árbol sintáctico no aparecen los paréntesis de la expresión. El árbol sintáctico ya indica el orden en que se procesan las diferentes operaciones y no necesita paréntesis. La expresión Python, sin embargo, *necesita* los paréntesis para indicar ese mismo orden de evaluación.

▶ 12 ¿Qué expresiones Python permiten, utilizando el menor número posible de paréntesis, efectuar *en el mismo orden* los cálculos representados con estos árboles sintácticos?







▶ 13 Dibuja los árboles sintácticos correspondientes a las siguientes expresiones aritméticas:

a) `1 + 2 + 3 + 4`

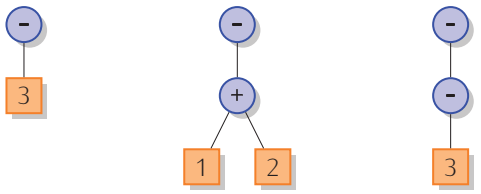
b) `1 - 2 - 3 - 4`

c) `1 - (2 - (3 - 4) + 1)`

Los operadores de suma y resta son *binarios*, es decir, operan sobre dos operandos. El mismo símbolo que se usa para la resta se usa también para un operador *unario*, es decir, un operador que actúa sobre un único operando: el de cambio de signo, que devuelve el valor de su operando cambiado de signo. He aquí algunos ejemplos:

```
>>> -3↵
-3
>>> -(1 + 2)↵
-3
>>> -3↵
3
```

He aquí los árboles sintácticos correspondientes a las tres expresiones del ejemplo:



Espacios en blanco

Parece que se puede hacer un uso bastante liberal de los espacios en blanco en una expresión.

```
>>> 10 + 20 + 30↵
60
>>> 10+20+30↵
60
>>> 10    +20   + 30↵
60
>>> 10+ 20+30↵
60
```

Es así. Has de respetar, no obstante, un par de sencillas reglas. Por una parte no puedes poner espacios en medio de un número:

```
>>> 10 + 2   0 + 30↵
  File "<stdin>", line 1
    10 + 2   0 + 30
             ^
SyntaxError: invalid syntax
```

Los espacios en blanco entre el 2 y el 0 hacen que Python no lea el número 20, sino el número 2 seguido del número 0 (lo cual es un error, pues no hay operación alguna entre ambos números).

Por otra parte, no puedes poner espacios al principio de la expresión:

```
>>>     10 + 20 + 30↵
  File "<stdin>", line 1
    10 + 20 + 30
    ^
IndentationError: unexpected indent
```

Los espacios en blanco entre el *prompt* y el 10 provocan un error. Aún es pronto para que conozcas la razón.

Existe otro operador unario que se representa con el símbolo +: el operador *identidad*. El operador identidad no hace nada «útil»: proporciona como resultado el mismo número que se le pasa.

```
>>> +3↵
3
>>> +-3↵
-3
```

El operador identidad solo sirve para, en ocasiones, poner énfasis en que un número es positivo. (El ordenador considera tan positivo el número 3 como el resultado de evaluar +3).

Los operadores de multiplicación y división son, respectivamente, * y /:

```
>>> 2 * 3↵
6
>>> 3 / 2↵
1.5
>>> 4 / 2↵
2.0
>>> 3 * 4 / 2↵
6.0
>>> 12 / 3 * 2↵
8.0
```

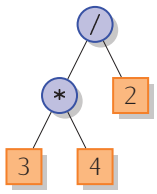
Detengámonos brevemente a hacer una consideración sobre el operador de división. Fíjate en que Python, al dividir 3 entre 2, ha proporcionado como respuesta el valor 1.5. Ese punto que separa el 1 del 5 es lo que en español solemos denotar con una coma[2] y que separa la parte entera de un número de su parte decimal. El operador de división / siempre proporciona un número con parte decimal, aunque esta sea nula; es lo que ocurre al dividir, por ejemplo, 4 entre 2 con el operador /: el resultado es 2.0. Hay otro operador de división que no tiene

[2] El punto también se acepta en español, aunque se prefiere usar la coma. Python representa los números siguiendo el convenio anglosajón.

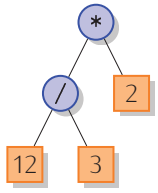
ese efecto: es el operador de división entera `//` (sin espacio alguno entre las barras). Con `//` siempre obtienes un número entero como resultado de la división de dos enteros:

```
>>> 3 // 2↵
1
>>> 4 // 2↵
2
>>> -3 // 2↵
-2
```

Observa que los operadores de multiplicación y división (convencional y entera) también son asociativos por la izquierda: la expresión «`3 * 4 / 2`» equivale a $((3 \cdot 4)/2)$, es decir, tiene el siguiente árbol sintáctico:



y la expresión «`12 / 3 * 2`» equivale a $((12/3) \cdot 2)$, o sea, su árbol sintáctico es:

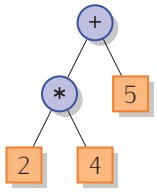


Sigamos estudiando el orden de evaluación cuando una expresión contiene diferentes operadores. ¿Qué pasa si combinamos en una misma expresión operadores de suma o resta con operadores de multiplicación o división? Fíjate en que la regla de aplicación de operadores de izquierda a derecha no siempre se observa:

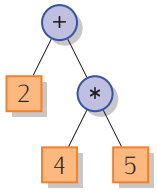
```
>>> 2 * 4 + 5↵
13
>>> 2 + 4 * 5↵
22
```

En la segunda expresión, primero se ha efectuado el producto `4 * 5` y el resultado se ha sumado al valor 2. Ocurre que los operadores de multiplicación y división son *prioritarios* frente a los de suma y resta. Decimos que la multiplicación y la división tienen *mayor nivel de precedencia* o *prioridad* que la suma y la resta.

El árbol sintáctico de `2 * 4 + 5` es:



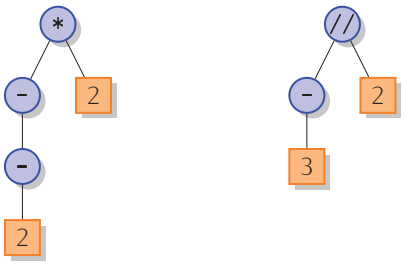
y el de `2 + 4 * 5` es:



Pero ¡atención!, el cambio de signo tiene mayor prioridad que la multiplicación y la división:

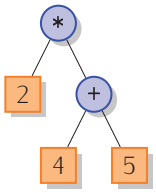
```
>>> -2 * 2↵
4
>>> -3 // 2↵
-2
```

Los árboles sintácticos correspondientes a estas dos expresiones son, respectivamente:



Si los operadores siguen unas *reglas de precedencia* que determinan su orden de aplicación, ¿qué hacer cuando deseamos un orden de aplicación distinto? Usar paréntesis, como hacemos con la notación matemática convencional.

La expresión `2 * (4 + 5)`, por ejemplo, presenta este árbol sintáctico:



Existen más operadores en Python. Tenemos, por ejemplo, el operador módulo, que se denota con el símbolo de porcentaje **%** (aunque nada tiene que ver con el cálculo de porcentajes). El operador módulo devuelve el resto de la división entera entre dos operandos.

```
>>> 27 % 5↵
2
>>> 25 % 5↵
0
```

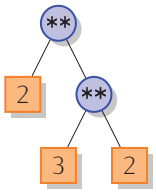
El operador **%** también es asociativo por la izquierda y su prioridad es la misma que la de la multiplicación o la división.

El último operador que vamos a estudiar es la exponenciación, que se denota con dos asteriscos juntos, no separados por ningún espacio en blanco: **.

Lo que en notación matemática convencional expresamos como 2^3 se expresa en Python con `2 ** 3`.

```
>>> 2 ** 3↵
8
```

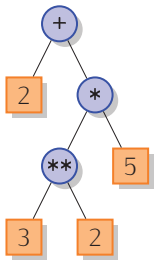
Pero ¡ojo!, la exponenciación es *asociativa por la derecha*. La expresión `2 ** 3 ** 2` equivale a $2^{(3^2)} = 2^9 = 512$, y no a $(2^3)^2 = 8^2 = 64$, o sea, su árbol sintáctico es:



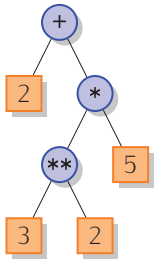
Por otra parte, la exponenciación tiene mayor precedencia que cualquiera de los otros operadores presentados.

He aquí varias expresiones evaluadas con Python y sus correspondientes árboles sintácticos. Estúdialos con atención:

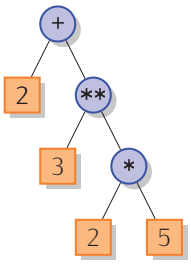
```
>>> 2 + 3 ** 2 * 5↵
47
```



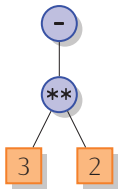

```
>>> 2 + ((3 ** 2) * 5)↵
47
```




```
>>> 2 + 3 ** (2 * 5)↵
59051
```




```
>>> -3 ** 2↵
-9
```



La tabla 2.1 resume las características de los operadores Python que ya conocemos: su aridad (número de operandos), asociatividad y precedencia.

▶ 14 ¿Qué resultados se obtendrán al evaluar las siguientes expresiones Python? Dibuja el árbol sintáctico de cada una de ellas, calcula a mano el valor resultante de cada expresión y comprueba, con la ayuda del ordenador, si tu resultado es correcto.

Operación	Operador	Aridad	Asociatividad	Precedencia
Exponenciación	**	Binario	Por la derecha	1
Identidad	+	Unario	—	2
Cambio de signo	-	Unario	—	2
Multiplicación	*	Binario	Por la izquierda	3
División	/	Binario	Por la izquierda	3
División entera	//	Binario	Por la izquierda	3
Módulo (o resto)	%	Binario	Por la izquierda	3
Suma	+	Binario	Por la izquierda	4
Resta	-	Binario	Por la izquierda	4

Tabla 2.1: Operadores para expresiones aritméticas. El nivel de precedencia 1 es el de mayor prioridad y el 4 el de menor.

a) `2 + 3 + 1 + 2`

b) `2 + 3 * 1 + 2`

c) `(2 + 3) * 1 + 2`

d) `(2 + 3) * (1 + 2)`

e) `+---6`

f) `-+-+6`

g) `-3 / 2 - 1`

h) `-3 // 2 - 1`

▶ 15 Traduce las siguientes expresiones matemáticas a Python y evalúalas. Trata de utilizar el menor número posible de paréntesis.

a) $2 + (3 \cdot (6/2))$

b) $\dfrac{4+6}{2+3}$

c) $(4/2)^5$

d) $(4/2)^{4+2^2}$

e) $(-3)^2$

f) $-(3^2)$

2.1.2. Errores de tecleo y excepciones

Cuando introducimos una expresión y damos la orden de evaluarla, es posible que nos equivoquemos. Si hemos formado incorrectamente una expresión, Python nos lo indicará con un *mensaje de error*. El mensaje de error proporciona información acerca del tipo de error cometido y del lugar en el que este ha sido detectado. Aquí tienes una expresión errónea y el mensaje de error correspondiente:

```
>>> 1 + 2)↵
  File "<stdin>", line 1
    1 + 2)
        ^
SyntaxError: invalid syntax
```

En este ejemplo hemos cerrado un paréntesis cuando no había otro abierto previamente, lo cual es incorrecto. Python nos indica que ha detectado un *error de sintaxis* (`SyntaxError`) y «apunta» con una punta de flecha (el carácter `^`) al lugar en el que se encuentra. (El texto «`File "<stdin>", line 1`» indica que el error se ha producido al leer de teclado, esto es, de la *entrada estándar* —`stdin` es una abreviatura del inglés «*standard input*», que se traduce por «entrada estándar»—).

En Python los errores se denominan *excepciones*. Cuando Python es incapaz de analizar una expresión, produce una excepción. Cuando el intérprete interactivo detecta la excepción, nos muestra por pantalla un mensaje de error.

Veamos algunos otros errores y los mensajes que produce Python.

```
>>> 1 + * 3↵
  File "<stdin>", line 1
    1 + * 3
        ^
SyntaxError: invalid syntax
>>> 2 + 3 %↵
  File "<stdin>", line 1
    2 + 3 %
          ^
SyntaxError: invalid syntax
>>> 1 / 0↵
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
```

El último error es de naturaleza distinta a los anteriores (no hay un carácter `^` apuntando a lugar alguno): se trata de un *error de división por cero* (`ZeroDivisionError`), cuando los otros eran *errores sintácticos* (`SyntaxError`). La cantidad que resulta de dividir por cero no está definida y Python es incapaz de calcular un valor como resultado de la expresión `1 / 0`. No es un error *sintáctico* porque la expresión está sintácticamente bien formada: el operador de división tiene dos operandos, como toca.

Edición avanzada en el entorno interactivo

Cuando estemos escribiendo una expresión puede que cometamos errores y los detectemos antes de solicitar su evaluación. Aún estaremos a tiempo de corregirlos. La tecla de borrado, por ejemplo, elimina el carácter que se encuentra a la izquierda del cursor. Puedes desplazar el cursor a cualquier punto de la línea que estás editando utilizando las teclas de desplazamiento del cursor a izquierda y a derecha. El texto que teclees se insertará siempre justo a la izquierda del cursor.

Hasta el momento hemos tenido que teclear desde cero cada expresión evaluada, aun cuando muchas se parecían bastante entre sí. Podemos teclear menos si aprendemos a utilizar algunas funciones de edición avanzadas.

Lo primero que hemos de saber es que el intérprete interactivo de Python memoriza cada una de las expresiones evaluadas en una sesión interactiva por si deseamos recuperarlas más tarde. La lista de expresiones que hemos evaluado constituye la *historia* de la sesión interactiva. Puedes «navegar» por la historia utilizando las teclas de desplazamiento del cursor hacia arriba y hacia abajo. Cada vez que pulses la tecla de desplazamiento hacia arriba recuperarás una expresión más antigua. La tecla de desplazamiento hacia abajo permite recuperar expresiones más recientes. La expresión recuperada aparecerá ante el *prompt* y podrás modificarla a tu antojo.

2.2. Tipos de datos

Ya hemos visto que hay dos operadores de división. Uno es el convencional, que siempre produce un número con decimales:

```
>>> 4 / 2↵
2.0
>>> 3 / 2↵
1.5
```

Y otro es el operador de división entera, que siempre produce un número sin decimales cuando sus operandos son enteros:

```
>>> 4 // 2↵
2
>>> 3 // 2↵
1
```

Fíjate en la diferencia entre el resultado de `4 / 2` y `4 // 2`. ¿No es lo mismo 2.0 que 2? Pues no exactamente. El número 2 es un número de *tipo entero* y el número 2.0 lo es de *tipo flotante* (o, mejor dicho, *tipo de coma flotante*). Cada valor en Python es una instancia de un *tipo de dato* y de momento hemos visto datos de dos tipos distintos.

2.2.1. Tipos entero y flotante

Hasta el momento hemos utilizado fundamentalmente datos de *tipo entero*, es decir, sin decimales. Solo ocasionalmente hemos visto datos de tipo *flotante*, normalmente como resultado de trabajar con el operador de división convencional. Pero no solo podemos producir datos de tipo flotante con el operador de división convencional: el resto de operadores produce un resultado cuyo tipo depende del tipo de sus operandos. La suma, por ejemplo, produce un resultado de tipo entero si sus dos operandos son de tipo entero, pero produce un valor de tipo flotante si uno cualquiera de sus operandos es de tipo flotante:

```
>>> 2 + 3↵
5
>>> 2.0 + 3↵
5.0
>>> 2 + 3.0↵
5.0
>>> 2.0 + 3.0↵
5.0
```

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
```

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 matemáticamente, pero 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, que debe ser entero, 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, 2000.0.

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 mayor o igual que 1 y menor que 10. El mismo número de antes, en coma flotante normalizada, es -1.41×10^{-2}. Una notación habitual para los números en coma flotante sustituye el producto ($\times$) 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 mayor o igual que 1 y 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. Un número como 0.1 no puede representarse exactamente como flotante. Su mantisa, que vale 1/10, corresponde a la secuencia periódica de bits

0.0001100110011001100110011001100110011001100110011001100110011

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.

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».

2.2.2. El tipo de datos booleano (y sus operadores)

Hay otro tipo de datos que has de conocer, pues se usa muy frecuentemente en programación: el tipo de datos lógico o booleano, llamado así por ser propio del álgebra de Boole[3]. Un dato de tipo lógico solo puede presentar uno de dos valores: **True** o **False**, es decir, verdadero o falso.

Hay tres operadores lógicos en Python: la «*y lógica*» o *conjunción* (**and**), la «*o lógica*» o *disyunción* (**or**) y el «*no lógico*» o *negación* (**not**).

El operador **and** da como resultado **True** si y solo si son **True** sus dos operandos. Esta es su *tabla de verdad*:

[3] Boole es el matemático que inventó (¿o descubrió?) el álgebra que lleva su nombre y que se basa en el uso de dos valores (cierto y falso) y tres operadores (negación, conjunción y disyunción).

and		
operandos		resultado
izquierdo	derecho	
True	True	True
True	False	False
False	True	False
False	False	False

El operador **or** proporciona **True** si cualquiera de sus operandos es **True**, y **False** solo cuando ambos operandos son **False**. Esta es su tabla de verdad:

or		
operandos		resultado
izquierdo	derecho	
True	True	True
True	False	True
False	True	True
False	False	False

El operador **not** es unario, y proporciona el valor **True** si su operando es **False** y viceversa. He aquí su tabla de verdad:

not	
operando	resultado
True	False
False	True

Podemos combinar valores lógicos y operadores lógicos para formar *expresiones lógicas*. He aquí algunos ejemplos:

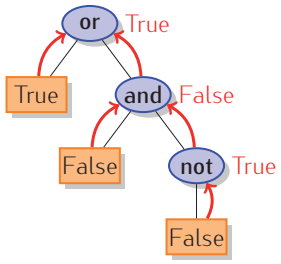
```
>>> True and False↵
False
>>> not True↵
False
>>> (True and False) or True↵
True
>>> True and True or False↵
True
>>> False and True or True↵
True
>>> False and True or False↵
False
```

Has de tener en cuenta la precedencia de los operadores lógicos, que se muestra en la tabla 2.2.

Operación	Operador	Aridad	Asociatividad	Precedencia
Negación	**not**	Unario	—	alta
Conjunción	**and**	Binario	Por la izquierda	media
Disyunción	**or**	Binario	Por la izquierda	baja

Tabla 2.2: Aridad, asociatividad y precedencia de los operadores lógicos.

Del mismo modo que hemos usado árboles sintácticos para entender el proceso de cálculo de los operadores aritméticos sobre valores enteros y flotantes, podemos recurrir a ellos para interpretar el orden de evaluación de las expresiones lógicas. He aquí el árbol sintáctico de la expresión **True or False and not False**:



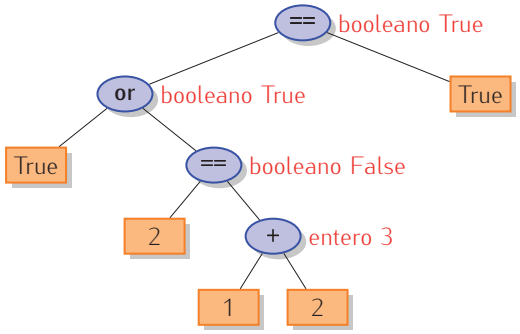
Hay una familia de operadores que devuelven valores booleanos. Entre ellos tenemos a los operadores de comparación, que estudiamos en este apartado. Uno de ellos es el operador de igualdad, que devuelve **True** si los valores comparados son iguales. El operador de igualdad se denota con dos iguales seguidos: **==**. Veámoslo en funcionamiento:

```
>>> 2 == 3↵
False
>>> 2 == 2↵
True
>>> 2.1 == 2.1↵
True
>>> True == True↵
True
>>> True == False↵
False
>>> 2 == 1+1↵
True
```

Observa la última expresión evaluada: es posible combinar operadores de comparación y operadores aritméticos. No solo eso, también podemos combinar en una misma expresión operadores lógicos, aritméticos y de comparación:

```
>>> (True or (2 == 1 + 2)) == True↵
True
```

Este es el árbol sintáctico correspondiente a esa expresión:



Hemos indicado junto a cada nodo interior el tipo del resultado que corresponde a su subárbol. Como ves, en todo momento operamos con tipos compatibles entre sí.

Antes de presentar las reglas de asociatividad y precedencia que son de aplicación al combinar diferentes tipos de operador, te presentamos todos los operadores de comparación en la tabla 2.3 y te mostramos algunos ejemplos de uso:

```
>>> 2 < 1↵
False
>>> 1 < 2↵
True
>>> 5 > 1↵
True
>>> 5 >= 1↵
True
>>> 5 > 5↵
False
```

operador	comparación
!=	es distinto de
==	es igual que
<	es menor que
<=	es menor o igual que
>	es mayor que
>=	es mayor o igual que

Tabla 2.3: Operadores de comparación.

```
>>> 5 >= 5↵
True
>>> 1 != 0↵
True
>>> 1 != 1↵
False
>>> -2 <= 2↵
True
```

Es hora de que presentemos una tabla completa (tabla 2.4) con todos los operadores que conocemos para comparar entre sí la precedencia de cada uno de ellos cuando aparece combinado con otros.

Operación	Operador	Aridad	Asociatividad	Precedencia
Exponenciación	**	Binario	Por la derecha	1
Identidad	+	Unario	—	2
Cambio de signo	-	Unario	—	2
Multiplicación	*	Binario	Por la izquierda	3
División	/	Binario	Por la izquierda	3
División entera	//	Binario	Por la izquierda	3
Módulo (o resto)	%	Binario	Por la izquierda	3
Suma	+	Binario	Por la izquierda	4
Resta	-	Binario	Por la izquierda	4
Igual que	==	Binario	—	5
Distinto de	!=	Binario	—	5
Menor que	<	Binario	—	5
Menor o igual que	<=	Binario	—	5
Mayor que	>	Binario	—	5
Mayor o Igual que	>=	Binario	—	5
Negación	not	Unario	—	6
Conjunción	and	Binario	Por la izquierda	7
Disyunción	or	Binario	Por la izquierda	8

Tabla 2.4: Características de los operadores Python. El nivel de precedencia 1 es el de mayor prioridad.

En la tabla 2.4 hemos omitido cualquier referencia a la asociatividad de los comparadores de Python, pese a que son binarios. Python es un lenguaje peculiar en este sentido. Imaginemos que fueran asociativos por la izquierda. ¿Qué significaría esto? El operador suma, por ejemplo, es asociativo por la izquierda. Al evaluar la expresión aritmética 2 + 3 + 4 se procede así: primero se suma el 2 al 3; a continuación, el 5 resultante se suma al 4 y se obtiene un total de 9.

Si el operador < fuese asociativo por la izquierda, la expresión lógica 2 < 3 < 4 se evaluaría así: primero, se compara el 2 con el 3, resultando el valor True; a continuación, se compara el resultado obtenido con el 4, pero ¿qué significa la expresión True < 4? No tiene sentido.

Cuando aparece una sucesión de comparadores como, por ejemplo, 2 < 3 < 4, Python la evalúa igual que (2 < 3) and (3 < 4). Esta solución permite expresar condiciones complejas de modo sencillo y, en casos como el de este ejemplo, se lee del mismo modo que se leería con la notación matemática habitual, lo cual parece deseable. Pero ¡ojo! Python permite expresiones que son más extrañas; por ejemplo, 2 < 3 > 1, o 2 < 3 == 5.

Una rareza de Python: la asociatividad de los comparadores

Algunos lenguajes de programación de uso común, como C y C++, hacen que sus operadores de comparación sean asociativos, por lo que presentan el problema de que expresiones como `2 < 1 < 4` producen un resultado que parece ilógico. Al ser asociativo por la izquierda el operador de comparación <, se evalúa primero la subexpresión `2 < 1`. El resultado es *falso*, que en C y C++ se representa con el valor 0. A continuación se evalúa la comparación `0 < 4`, cuyo resultado es... ¡cierto! Así pues, para C y C++ es cierto que `2 < 1 < 4`.

Pascal es más rígido aún y llega a prohibir expresiones como `2 < 1 < 4`. En Pascal hay un tipo de datos denominado `boolean` cuyos valores válidos son `true` y `false`. Pascal no permite operar entre valores de tipos diferentes, así que la expresión `2 < 1` se evalúa al valor booleano `false`, que no se puede comparar con un entero al tratar de calcular el valor de `false < 4`. En consecuencia, se produce un error de tipos si intentamos encadenar comparaciones.

La mayor parte de los lenguajes de programación convencionales opta por la solución del C o por la solución del Pascal. Cuando aprendas otro lenguaje de programación, te costará «deshabituarte» de la elegancia con que Python resuelve los encadenamientos de comparaciones.

▶ 16 ¿Qué resultados se muestran al evaluar estas expresiones?

```
>>> True == True != False↵
>>> 1 < 2 < 3 < 4 < 5↵
>>> (1 < 2 < 3) and (4 < 5)↵
>>> 1 < 2 < 4 < 3 < 5↵
>>> (1 < 2 < 4) and (3 < 5)↵
```

2.3. Literales de entero

Ya sabes cómo escribir un número entero... en base 10. Es lo más corriente: escribir números enteros con un sistema en el que cada dígito es un número entre 0 y 9, y en el que cada posición supone que el dígito vale 10 veces más que el que hay más a su derecha. Así, el número 132 equivale a $1 \cdot 10^2 + 3 \cdot 10^1 + 2 \cdot 10^0 = 1 \cdot 100 + 3 \cdot 10 + 2 \cdot 1$. Pero en ocasiones convendrá expresar números con otra base. En informática es muy común trabajar con base 2, base 8 y base 16. Es natural que la base 2 sea usada con relativa frecuencia, pues ya sabes que la información se almacena y manipula a partir de codificaciones binarias. Las otras dos bases son de uso común por razones históricas. La base 8 permite manejar números en los que interpretamos que cada grupo de 3 bits corresponde a un dígito y la base 16 hace lo propio con grupos de 4 bits.

Python permite expresar números en estas tres bases. Los números en base 2 se expresan con el prefijo 0b (o 0B) seguido de una sucesión de unos y ceros.

```
>>> 0b1 + 0b0001↵
2
>>> 0b10 * 0b110↵
12
```

Observa que el resultado se muestra en base 10, aunque los números se hayan escrito en base 2. Tan pronto Python analiza la secuencia 0b10, por ejemplo, considera que ha leído el valor 2. Internamente no hay ninguna diferencia entre el resultado de evaluar 2 y 0b10. Así pues, el resultado de evaluar la expresión 0b1 + 0b0001 es exactamente el mismo que obtenemos al evaluar 1+1.

Los números en base 8, también conocida como base octal, se forman con 0o (o 0O) seguida de uno o más dígitos entre 0 y 7. El número 0o177 corresponde al número decimal $1 \cdot 8^2 + 7 \cdot 8^1 + 7 \cdot 8^0 = 1 \cdot 64 + 7 \cdot 8 + 7 \cdot 1 = 64 + 56 + 7 = 127$.

```
>>> 0o10 + 0o10↵
16
>>> 0o7 + 0o177↵
134
```

Finalmente, los números en base 16 o hexadecimal empiezan por 0x (o 0X) y siguen con uno o más dígitos entre 0 y 9 o letras minúsculas o mayúsculas entre A y F. La letra A tiene valor 10, la B valor 11 y así hasta la F, que tiene valor 15. El número 0xFA1 corresponde a $15 \cdot 16^2 + 10 \cdot 16^1 + 1 \cdot 16^0 = 15 \cdot 256 + 10 \cdot 16 + 1 \cdot 1 = 4001$.

```
>>> 0xff + 0x1↵
256
>>> 0xff * 0x2↵
510
```

¿Y si quisiésemos ver el resultado en base 2, 8 o 16? Hemos de aprender antes algunas cosas acerca de las cadenas y las funciones. Antes de acabar este capítulo sabremos cómo hacerlo.

▶ 17 Evalúa estas expresiones:

a) 0xf + 0o17 + 0b1111 + 15

b) 0xffff + 0b1

2.4. Variables y asignaciones

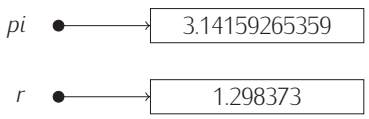
En ocasiones deseamos que el ordenador recuerde ciertos valores para usarlos más adelante. Por ejemplo, supongamos que deseamos efectuar el cálculo del perímetro y el área de un círculo de radio 1.298373 m. La fórmula del perímetro es $2\pi r$, donde r es el radio, y la fórmula del área es πr^2. (Aproximaremos el valor de π con 3.14159265359). Podemos realizar ambos cálculos del siguiente modo:

```
>>> 2 * 3.14159265359 * 1.298373↵
8.157918156839218
>>> 3.14159265359 * 1.298373 ** 2↵
5.296010335524904
```

Observa que hemos tenido que introducir dos veces los valores de π y r por lo que, al tener tantos decimales, es muy fácil cometer errores. Para paliar este problema podemos utilizar *variables*:

```
>>> pi = 3.14159265359↵
>>> r = 1.298373↵
>>> 2 * pi * r↵
8.157918156839218
>>> pi * r ** 2↵
5.296010335524904
```

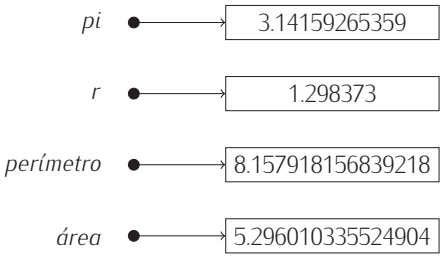
En la primera línea hemos creado una variable de nombre *pi* y valor 3.14159265359. A continuación, hemos creado otra variable, *r*, y le hemos dado el valor 1.298373. El acto de dar valor a una variable se denomina *asignación*. Al asignar un valor a una variable que no existía, Python reserva un espacio en la memoria, almacena el valor en él y crea una asociación entre el nombre de la variable y la dirección de memoria de dicho espacio. Podemos representar gráficamente el resultado de estas acciones así:



A partir de ese instante, escribir *pi* es equivalente a escribir 3.14159265359, y escribir *r* es equivalente a escribir 1.298373.

Podemos almacenar el resultado de calcular el perímetro y el área en sendas variables:

```
>>> pi = 3.14159265359↵
>>> r = 1.298373↵
>>> perímetro = 2 * pi * r↵
>>> área = pi * r**2↵
```



La memoria se ha reservado correctamente, en ella se ha almacenado el valor correspondiente y la asociación entre la memoria y el nombre de la variable se ha establecido, pero no obtenemos respuesta alguna por pantalla. Debes tener en cuenta que las asignaciones son «mudas», es decir, no provocan salida por pantalla. Si deseamos ver cuánto vale una variable, podemos evaluar una expresión que solo contiene a dicha variable:

```
>>> pi = 3.14159265359↵
>>> r = 1.298373↵
>>> perímetro = 2 * pi * r↵
>>> área = pi * r**2↵
>>> perímetro↵
8.157918156839218
>>> área↵
5.296010335524904
```

Así pues, para asignar valor a una variable basta ejecutar una sentencia como esta:

variable = *expresión*

Ten cuidado: el orden es importante. Hacer «*expresión* = *variable*» no es equivalente. *Una asignación no es una ecuación matemática*, sino una acción consistente en (por este orden):

1) evaluar la expresión *a la derecha* del símbolo igual (=), y

2) guardar el valor resultante en la variable indicada *a la izquierda* del símbolo igual.

== no es = (comparar no es asignar)

Al aprender a programar, muchas personas confunden el operador de asignación, =, con el operador de comparación, ==. El primero se usa exclusivamente para asignar un valor a una variable. El segundo, para comparar valores.

Observa la diferente respuesta que obtienes al usar = y == en el entorno interactivo:

```
>>> a = 10↵
>>> a↵
10
>>> a == 1↵
False
>>> a↵
10
```

Se puede asignar valor a una misma variable cuantas veces se quiera. El efecto es que la variable, en cada instante, solo «recuerda» el último valor asignado... hasta que se le asigne otro.

```
>>> a = 1↵
>>> 2 * a↵
2
>>> a + 2↵
3
>>> a = 2↵
>>> a * a↵
4
```

Una asignación no es una ecuación

Hemos de insistir en que las asignaciones no son ecuaciones matemáticas, por mucho que su aspecto nos recuerde a estas. Fíjate en este ejemplo, que suele sorprender a aquellos que empiezan a programar:

```
>>> x = 3↵
>>> x = x + 1↵
>>> x↵
4
```

La primera línea asigna a la variable *x* el valor 3. La segunda línea parece más complicada. Si la interpretas como una ecuación, no tiene sentido, pues de ella se concluye absurdamente que $3 = 4$ o, sustrayendo la *x* a ambos lados del igual, que $0 = 1$. Pero si seguimos paso a paso las acciones que ejecuta Python al hacer una asignación, la cosa cambia:

1) Se evalúa la parte derecha del igual (sin tener en cuenta para nada la parte izquierda). El valor de *x* es 3, que sumado a 1 da 4.
2) El resultado (el 4), se almacena en la variable que aparece en la parte izquierda del igual, es decir, en *x*.

Así pues, el resultado de ejecutar las dos primeras líneas es que *x* vale 4.

El nombre de una variable es su *identificador*. Hay unas reglas precisas para construir identificadores. Si no se siguen, diremos que el identificador no es válido. Un identificador debe estar formado por letras[4] minúsculas, mayúsculas, dígitos y/o el carácter de subrayado _, con una restricción: que el primer carácter no sea un dígito.

Hay una norma más: un identificador no puede coincidir con una *palabra reservada* o *palabra clave*. Una palabra reservada es una palabra que tiene un significado predefinido y es necesaria para expresar ciertas construcciones del lenguaje. Aquí tienes una lista con todas las palabras reservadas de Python: **and**, **as**, **assert**, **break**, **class**, **continue**, **def**, **del**, **elif**, **else**, **except**, **False**, **finally**, **for**, **from**, **global**, **if**, **import**, **in**, **is**, **lambda**, **nonlocal**, **None**, **not**, **or**, **pass**, **raise**, **return**, **True**, **try**, **with**, **while** y **yield**.

Por ejemplo, los siguientes identificadores son válidos: *h*, *x*, *Z*, *velocidad*, *aceleración*, *fuerza1*, *masa_2*, *_a*, *a_*, *prueba_123*, *desviación_típica*. Debes tener presente que Python distingue entre mayúsculas y minúsculas, así que *área*, *Área* y *ÁREA* son tres identificadores válidos y diferentes.

Cualquier carácter diferente de una letra, un dígito o el subrayado es inválido en un identificador, incluyendo el espacio en blanco. Por ejemplo, *edad media* (con un espacio en medio) son *dos* identificadores (*edad* y *media*), no uno. Cuando un identificador se forma con dos palabras, es costumbre de muchos programadores usar el subrayado para separarlas: *edad_media*; otros programadores utilizan una letra mayúscula para la inicial de la segunda: *edadMedia*. Escoge el estilo que más te guste para nombrar variables, pero permanece fiel al que escojas.

Dado que eres libre de llamar a una variable con el identificador que quieras, hazlo con clase: *escoge siempre nombres que guarden relación con los datos del problema*. Si, por ejemplo, vas a utilizar una variable para almacenar una distancia, llama a la variable *distancia* y evita nombres que no signifiquen nada; de este modo, los programas serán más legibles.

▶ 18 ¿Son válidos los siguientes identificadores?

a) *Identificador*

[4] Son letras válidas las de cualquier alfabeto. Así, π es un identificador válido por ser una letra del alfabeto griego.

b) *Indice\dos*

c) *Dos palabras*

d) __

e) 12*horas*

f) *hora12*

g) *desviación*

h) *año*

i) *from*

j) *var*!

k) `'var'`

l) *import_from*

m) *UnaVariable*

n) *a*(*b*)

ñ) 12

o) *uno*.*dos*

p) *x*

q) π

r) *área*

s) *area*-*rect*

t) *x_____1*

u) *_____1*

v) *_x_*

w) *x_x*

▶ 19 ¿Qué resulta de ejecutar estas tres líneas?

```
>>> x = 10↵
>>> x = x * 10↵
>>> x↵
```

▶ 20 Evalúa el polinomio $x^4 + x^3 + 2x^2 - x$ en $x = 1.1$. Utiliza variables para evitar teclear varias veces el valor de x. (El resultado es 4.1151).

▶ 21 Evalúa el polinomio $x^4 + x^3 + \frac{1}{2}x^2 - x$ en $x = 10$. (El resultado es 11040.0).

2.4.1. Asignaciones con operador

Fíjate en la sentencia *i* = *i* + 1: aplica un incremento unitario al contenido de la variable *i*. Incrementar el valor de una variable en una cantidad cualquiera es tan frecuente que existe una forma compacta en Python. El incremento de *i* puede denotarse así: *i* += 1 (sin espacio alguno entre el + y el =). Puedes incrementar una variable con cualquier cantidad, incluso con una que resulte de evaluar una expresión:

```
>>> a = 3↵
>>> b = 2↵
>>> a += 4 * b↵
>>> a↵
11
```

Todos los operadores aritméticos tienen su asignación con operador asociada.

```
>>> z = 1↵
>>> z += 2↵
>>> z *= 2↵
>>> z //= 2↵
>>> z -= 2↵
>>> z %= 2↵
>>> z **= 2↵
>>> z /= 2↵
>>> z↵
0.5
```

Hemos de decirte que estas formas compactas no aportan nada nuevo... salvo comodidad, así que no te preocupes por tener que aprender tantas cosas. Si te vas a sentir incómodo por tener que tomar decisiones y siempre estás pensando «¿uso ahora la forma normal o la compacta?», será mejor que ignores de momento las formas compactas.

▶ 22 ¿Qué valor tiene *z* tras evaluar estas sentencias?

```
>>> z = 2↵
>>> z += 2↵
>>> z += 2 - 2↵
>>> z *= 2↵
>>> z *= 1 + 1↵
>>> z //= 2↵
>>> z %= 3↵
>>> z /= 3 - 1↵
>>> z -= 2 + 1↵
>>> z -= 2↵
>>> z **= 3↵
>>> z↵
```

2.4.2. Variables no inicializadas

En Python, la primera operación sobre una variable debe ser la asignación de un valor. No se puede usar una variable a la que no se ha asignado previamente un valor:

```
>>> a + 2↵
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'a' is not defined
```

Como puedes ver, se genera una excepción `NameError`, es decir, de «error de nombre». El texto explicativo precisa aún más lo sucedido: «`name 'a' is not defined`», es decir, «el nombre *a* no está definido».

La asignación de un valor inicial a una variable se denomina *inicialización* de la variable. Decimos, pues, que en Python no es posible usar variables no inicializadas.

Más operadores

Solo te hemos presentado los operadores que utilizaremos en el texto y que ya estás preparado para manejar. Pero has de saber que hay más operadores. Hay operadores, por ejemplo, que están dirigidos a manejar las secuencias de bits que codifican los valores enteros. El operador binario **&** calcula la operación «y» bit a bit, el operador binario **|** calcula la operación «o» bit a bit, el operador binario **^** calcula la «o exclusiva» (que devuelve cierto si y solo si los dos operandos son distintos), también bit a bit, y el operador unario **~** invierte los bits de su operando. Tienes, además, los operadores binarios **<<** y **>>**, que desplazan los bits a izquierda o derecha tantas posiciones como le indiques. Estos ejemplos te ayudarán a entender estos operadores:

En decimal		En binario	
Expresión	Resultado	Expresión	Resultado
5 & 12	4	00000101 & 00001100	00000100
5 | 12	13	00000101 | 00001100	00001101
5 ^ 12	9	00000101 ^ 00001100	00001001
5 << 1	10	00000101 << 00000001	00001010
5 << 2	20	00000101 << 00000010	00010100
5 << 3	40	00000101 << 00000011	00101000
5 >> 1	2	00000101 >> 00000001	00000010

¡Y estos operadores presentan, además, una forma compacta con asignación: **<<=**, **|=**, etc.!

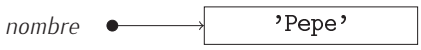
Más adelante estudiaremos, además, los operadores **is** (e **is not**) e **in** (y **not in**), los operadores de indexación, de llamada a función, de corte...

2.5. El tipo de datos cadena

Hasta el momento hemos visto que Python puede manipular datos numéricos de dos tipos: enteros y flotantes. Pero Python también puede manipular otros tipos de datos. Vamos a estudiar ahora el tipo de datos que se denomina *cadena*. Una cadena es una secuencia de caracteres (letras, números, espacios, marcas de puntuación, etc.) y en Python se distingue porque va *encerrada entre comillas simples o dobles* . Por ejemplo, `'cadena'`, `'otro␣ejemplo'`, `"1,␣2␣1o␣3"`, `'¡Si!'`, `"...Python"` son cadenas. Observa que los espacios en blanco se muestran así en este texto: «␣». Lo hacemos para que resulte fácil contar los espacios en blanco cuando haya más de uno seguido. Esta cadena, por ejemplo, está formada por tres espacios en blanco: `'␣␣␣'`. Si no los representásemos con las cajitas, sería difícil contarlos.

Las cadenas pueden usarse para representar información textual: nombres de personas, nombres de colores, matrículas de coche... Las cadenas también pueden almacenarse en variables.

```
>>> nombre = 'Pepe'↵
>>> nombre↵
'Pepe'
```



Es posible realizar operaciones con cadenas. Por ejemplo, podemos «sumar» cadenas añadiendo una a otra.

```
>>> 'a' + 'b'↵
'ab'
>>> nombre = 'Pepe'↵
>>> nombre + 'Cano'↵
'PepeCano'
>>> nombre + '␣' + 'Cano'↵
'Pepe␣Cano'
>>> apellido = 'Cano'↵
>>> nombre + '␣' + apellido↵
'Pepe␣Cano'
```

Una cadena no es un identificador

Con las cadenas tenemos un problema: muchas personas que están aprendiendo a programar confunden una cadena con un identificador de variable y viceversa. No son la misma cosa. Fíjate bien en lo que ocurre:

```
>>> a = 1↵
>>> 'a'↵
'a'
>>> a↵
1
```

La primera línea asigna a la variable *a* el valor 1. Como *a* es el nombre de una variable, es decir, un identificador, *no va encerrado entre comillas*. A continuación hemos escrito `'a'` y Python ha respondido también con `'a'`: la *a* entre comillas es una cadena formada por un único carácter, la letra «a», y no tiene *nada* que ver con la variable *a*. A continuación hemos escrito la letra «a» sin comillas y Python ha respondido con el valor 1, que es lo que contiene la variable *a*.

Muchos estudiantes de programación cometen errores como estos:

- Quieren utilizar una cadena, pero olvidan las comillas, con lo que Python cree que se quiere usar un identificador; si ese identificador no existe, da un error:

```
>>> Pepe↵
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'Pepe' is not defined
```

- Quieren usar un identificador pero, ante la duda, lo encierran entre comillas:

```
>>> 'x' = 2↵
  File "<input>", line 1
SyntaxError: can't assign to literal
```

Recuerda: solo se puede asignar valores a variables, nunca a cadenas, y las cadenas no son identificadores.

Hablando con propiedad, esta operación no se llama suma, sino *concatenación*. El símbolo utilizado es `+`, el mismo que usamos cuando sumamos enteros y/o flotantes; pero aunque el símbolo sea el mismo, ten en cuenta que no es igual sumar números que concatenar cadenas:

```
>>> '12' + '12'↵
'1212'
>>> 12 + 12↵
24
```

Sumar o concatenar una cadena y un valor numérico (entero o flotante) produce un error:

```
>>> '12' + 12↵
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Y para acabar, hay un operador de repetición de cadenas. El símbolo que lo denota es `*`, el mismo que hemos usado para multiplicar enteros y/o flotantes. El operador de repetición necesita dos datos: uno de tipo cadena y otro de tipo entero. El resultado es la concatenación de la cadena consigo misma tantas veces como indique el número entero:

```
>>> 'Hola' * 5↵
'HolaHolaHolaHolaHola'
>>> '-' * 60↵
'------------------------------------------------------------'
>>> 60 * '-'↵
'------------------------------------------------------------'
```

▶ 23 Evalúa estas expresiones y sentencias en el mismo orden en el que aparecen e indica lo que muestra el intérprete de Python como respuesta.

```
>>> a = 'b'↵
>>> a + 'b'↵
>>> a + 'a'↵
>>> a * 2 + 'b' * 3↵
>>> 2 * (a + 'b')↵
>>> 2 * ('a' + 'b')↵
```

▶ 24 ¿Qué resultados se obtendrán al evaluar las siguientes expresiones y asignaciones Python? Calcula primero a mano el valor resultante de cada expresión y comprueba, con la ayuda del ordenador, si tu resultado es correcto.

```
>>> 'a' * 3 + '/*' * 5 + 2 * 'abc' + '+'↵
>>> palindromo = 'abcba'↵
>>> (4 * '<' + palindromo + '>' * 4) * 2↵
>>> subcadena = '=' + '-' * 3 + '='↵
>>> '10' * 5 + 4 * subcadena↵
>>> 2 * '12' + '.' + '3' * 3 + 'e-' + 4 * '76'↵
```

▶ 25 Identifica regularidades en las siguientes cadenas, y escribe expresiones que, partiendo de subcadenas más cortas y utilizando los operadores de concatenación y repetición, produzcan las cadenas que se muestran. Introduce variables para formar las expresiones cuando lo consideres oportuno.

a) `'%%%%%./././<->'`

b) `'(@)(@)(@)======(@)(@)(@)======'`

c) `'asdfasdfasdf=-=-=-=-=-=-=-??????asdfasdf'`

d) `'........*****--*****--........*****--*****--'`

2.6. Funciones predefinidas

Hemos estudiado los operadores aritméticos básicos. Python también proporciona funciones que podemos utilizar en las expresiones. Estas funciones se dice que están *predefinidas*[5].

2.6.1. Algunas funciones sobre valores numéricos

La función *abs*, por ejemplo, calcula el valor absoluto de un número. Podemos usarla como en estas expresiones:

```
>>> abs(-3)↵
3
>>> abs(3)↵
3
```

El número sobre el que se aplica la función se denomina *argumento*. Observa que el argumento de la función debe ir encerrado entre paréntesis:

```
>>> abs(0)↵
0
>>> abs 0↵
  File "<input>", line 1
    abs 0
        ^
SyntaxError: invalid syntax
```

Existen muchas funciones predefinidas, pero es pronto para aprenderlas todas. Te resumimos algunas que ya puedes utilizar:

[5] *Pre*definidas porque nosotros también podemos definir nuestras propias funciones. Ya llegaremos.

- *float*: conversión a flotante. Si recibe un número entero como argumento, devuelve el mismo número convertido en un flotante equivalente.

```
>>> float(3)↵
3.0
```

  La función *float* también acepta argumentos de tipo cadena. Cuando se le pasa una cadena, *float* la convierte en el número flotante que esta representa:

```
>>> float('3.2')↵
3.2
>>> float('3.2e10')↵
32000000000.0
```

  Pero si la cadena no representa un flotante, se produce un error de tipo `ValueError`, es decir, «error de valor»:

```
>>> float('un texto')↵
Traceback (most recent call last):
  File "<input>", line 1, in <module>
ValueError: could not convert string to float: 'un texto'
```

  Si *float* recibe un argumento flotante, devuelve el mismo valor que se suministra como argumento.

- *int*: conversión a entero. Si recibe un número flotante como argumento, devuelve el entero que se obtiene eliminando la parte fraccionaria.[6]

```
>>> int(2.1)↵
2
>>> int(-2.9)↵
-2
```

  La función *int* acepta como argumento una cadena:

```
>>> int('2')↵
2
```

  Si *int* recibe un argumento entero, devuelve un entero con el valor del argumento, tal cual.

- *str*: conversión a cadena. Recibe un número y devuelve una representación de este como cadena.

```
>>> str(2.1)↵
'2.1'
>>> str(234E47)↵
'2.34e+49'
```

  La función *str* también puede recibir como argumento una cadena, pero en ese caso devuelve como resultado la misma cadena.

[6] El redondeo de *int* puede ser al alza o a la baja según el ordenador en que lo ejecutes. Esto es así porque *int* se apoya en el comportamiento del redondeo automático de C (el intérprete de Python que usamos está escrito en C) y su comportamiento está indefinido. Si quieres un comportamiento homogéneo del redondeo, puedes usar las funciones *round*, *floor* o *ceil*, que se explican más adelante.

- *bin*: representación en binario. Convierte un número entero en una cadena con el número expresado en base 2.

```
>>> bin(3)↵
'0b11'
>>> bin(10)↵
'0b1010'
>>> bin(254)↵
'0b11111110'
```

- *oct*: representación en octal. Convierte un número entero en una cadena con el número expresado en base 8.

```
>>> oct(3)↵
'0o3'
>>> oct(10)↵
'0o12'
>>> oct(254)↵
'0o376'
```

- *hex*: representación en hexadecimal. Convierte un número entero en una cadena con el número expresado en base 16.

```
>>> hex(3)↵
'0x3'
>>> hex(10)↵
'0xa'
>>> hex(254)↵
'0xfe'
```

- *round*: redondeo. Puede usarse con uno o dos argumentos. Si se usa con un solo argumento, redondea el número al entero más próximo.

```
>>> round(2.1)↵
2
>>> round(2.9)↵
3
>>> round(-2.9)↵
-3
>>> round(2)↵
2
```

(¡Observa que el resultado siempre es de tipo entero!) Si *round* recibe dos argumentos, *estos deben ir separados por una coma* y el segundo indica el número de decimales que deseamos conservar tras el redondeo.

```
>>> round(2.1451, 2)↵
2.15
>>> round(2.1451, 3)↵
2.145
>>> round(2.1451, 0)↵
2.0
```

Estas funciones (y las que estudiaremos más adelante) pueden formar parte de expresiones y sus argumentos pueden, a su vez, ser expresiones. Observa los siguientes ejemplos:

```
>>> abs(-23) % int(7.3)↵
2
>>> abs(round(-34.2765, 1))↵
34.3
>>> str(float(str(2) * 3 + '.123')) + '321'↵
'222.123321'
```

▶ 26 Calcula con una única expresión el valor absoluto del redondeo de −3.2. (El resultado es 3).

▶ 27 Convierte (en una única expresión) a una cadena el resultado de la división 5011/10000 redondeado con 3 decimales.

▶ 28 ¿Qué resulta de evaluar estas expresiones?

```
>>> str(2.1) + str(1.2)↵
>>> int(str(2) + str(3))↵
>>> str(int(12.3)) + '0'↵
>>> int('2'+'3')↵
>>> str(2 + 3)↵
>>> str(int(2.1) + float(3))↵
```

2.6.2. Dos funciones básicas para cadenas: *ord* y *chr*

El concepto de comparación entre números te resulta familiar porque lo has estudiado antes en matemáticas. Python extiende el concepto de comparación a otros tipos de datos, como las cadenas. En el caso de los operadores == y != el significado está claro: dos cadenas son iguales si son iguales carácter a carácter, y distintas en caso contrario. Pero ¿qué significa que una cadena sea menor que otra? Python utiliza un criterio de comparación de cadenas similar al orden alfabético.

Cuando Python compara dos cadenas lo hace carácter a carácter. Cada carácter es un entero de 16 bits. La letra a, por ejemplo, se codifica con el entero 97, y la b con el entero 98. Si comparamos la cadena `'a'` con la cadena `'b'`, Python nos dirá que la primera es menor que la segunda. ¿Qué pasa si comparamos `'aa'` con `'ab'`? Python compara primero el primer carácter de cada cadena. Como los dos son iguales, pasa a comparar el segundo carácter de cada cadena y llega a la conclusión de que la primera cadena es menor que la segunda. ¿Y qué pasa si comparamos `'a'` con `'aa'`? Nuevamente el primer carácter resulta insuficiente para decidir nada. Python trata de pasar a estudiar el segundo carácter de cada cadena, pero la primera cadena no tiene segundo carácter. Así pues, nuevamente resulta que la primera cadena es menor que la segunda.

En principio, una cadena es menor que otra si la debe preceder al disponerlas en un diccionario. Por ejemplo, `'abajo'` es menor que `'arriba'`. Pero solo en principio. Fíjate en que la letra b mayúscula tiene código 66. Eso significa que 'Barco' es menor que 'ancla':

```
>>> 'Barco' < 'ancla'↵
True
```

Las letras acentuadas plantean problemas similares, pues tienen valores numéricos mayores que sus versiones sin acentuar:

```
>>> 'ábaco' < 'ajo'↵
False
```

Para conocer el valor numérico que corresponde a un carácter, puedes utilizar la función predefinida *ord*, a la que le has de pasar el carácter en cuestión como argumento.

```
>>> ord('a')↵
97
>>> ord('A')↵
65
```

De ASCII a Unicode

Hace algunos años era corriente codificar cada carácter con un byte (ocho bits). La tabla que establecía la correspondencia entre carácter y valor numérico era la denominada tabla ASCII, de la que hemos hablado brevemente en el primer capítulo. La letra a, por ejemplo, tiene valor numérico 97. En realidad no codificaba 256 símbolos, sino solo 128: los que correspondían a valores numéricos entre 0 y 127. Esta tabla, diseñada en 1968, era problemática en países como el nuestro, pues no recogía los caracteres acentuados o la letra eñe. Con 256 caracteres, que es lo que podemos codificar con 8 bits, era imposible tener un juego completo válido para todos los países del mundo. De hecho, ni siquiera para todos los países europeos.

Cada sistema informático extendió la tabla ASCII a su gusto. Usualmente se añadían caracteres a los 128 valores no usados por la tabla ASCII. Esto trajo multitud de problemas a la hora de intercambiar archivos de texto entre sistemas. En los años 90, una comisión estandarizó tablas adaptadas a diferentes dominios lingüísticos. La tabla apropiada para Europa occidental era la denominada ISO-8859-1, también conocida como IsoLatin1 o Latin1. Pronto esta tabla se quedó corta: el símbolo del euro no estaba contemplado en ella. La tabla ISO-8859-15 ampliaba la ISO-8859-1 para recoger el símbolo del euro.

El problema de codificar la información textual estaba lejos de quedar satisfactoriamente resuelto si había que recurrir a multitud de tablas de 256 caracteres. Piénsese en que era imposible, por ejemplo, incluir en un único fichero de texto un fragmento en español con otro en japonés.

Surgió entonces una codificación capaz de resolver el problema definitivamente: la codificación Unicode. Unicode empezó planteando que cada carácter debía codificarse con 16 bits y no con solo 8. Esto hacía que hubiera 65536 códigos disponibles. Como seguían siendo insuficientes para representar cualquier carácter de cualquier lengua, Unicode definió codificaciones con número de bits variable que permitieran, mediante sucesivas extensiones, dar cuenta de cualquier alfabeto existente.

Nosotros consideraremos que cada carácter se representa con un número de 16 bits y no entraremos en más detalles sobre Unicode. Python, desde las versiones 3.0 en adelante, representa las cadenas con Unicode.

```
>>> ord('á')↵
225
```

La función *chr* hace lo contrario: devuelve un carácter, dado su valor numérico.

```
>>> chr(97)↵
'a'
>>> chr(65)↵
'A'
```

▶ 29 ¿Qué resultados se muestran al evaluar estas expresiones?

```
>>> 'abalorio' < 'abecedario'↵
>>> 'abecedario' < 'abecedario'↵
>>> 'abecedario' <= 'abecedario'↵
>>> 'Abecedario' < 'abecedario'↵
>>> 'Abecedario' == 'abecedario'↵
>>> 124 < 13↵
>>> '124' < '13'↵
>>> '␣a' < 'a'↵
```

2.7. Módulos e importación de funciones y variables

Python también proporciona funciones trigonométricas, logaritmos, etc., pero no están directamente disponibles cuando iniciamos una sesión. Antes de utilizarlas hemos de indicar a Python que vamos a hacerlo. Para ello, *importamos* cada función de un módulo.

2.7.1. El módulo *math*

Empezaremos por importar la función seno (*sin*, del inglés «sinus») del módulo matemático (*math*):

```
>>> from math import sin↵
```

Ahora podemos utilizar la función en nuestros cálculos:

```
>>> from math import sin↵
>>> sin(0)↵
0.0
>>> sin(1)↵
0.8414709848078965
```

Observa que el argumento de la función seno debe expresarse en radianes.

Inicialmente Python no «sabe» calcular la función seno. Cuando importamos una función, Python «aprende» su definición y nos permite utilizarla. Las definiciones de funciones residen en *módulos*. Las funciones trigonométricas residen en el módulo matemático. Por ejemplo, la función coseno, en este momento, es desconocida para Python.

```
>>> cos(0)↵
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'cos' is not defined
```

Antes de usarla, es necesario importarla del módulo matemático:

```
>>> from math import cos↵
>>> cos(0)↵
1.0
```

En una misma sentencia podemos importar más de una función. Basta con separar sus nombres con comas:

```
>>> from math import sin, cos↵
```

Puede resultar tedioso importar un gran número de funciones y variables de un módulo. Python ofrece un atajo: si utilizamos un asterisco, se importan *todos* los elementos proporcionados por un módulo. Para importar todos los elementos del módulo *math* escribimos:

```
>>> from math import *↵
```

Así de fácil. De todos modos, no resulta muy aconsejable por dos razones:

- Al importar elemento a elemento, el programa gana en legibilidad, pues sabemos de dónde proviene cada identificador.
- Si hemos definido una variable con un nombre determinado y dicho nombre coincide con el de una función definida en un módulo, nuestra variable será sustituida por la función. Si no sabes todos los elementos que define un módulo, es posible que esta coincidencia de nombre tenga lugar, te pase inadvertida inicialmente y te lleves una sorpresa cuando intentes usar la variable.

He aquí un ejemplo del segundo de los problemas indicados:

```
>>> pow = 1↵
>>> from math import *↵
>>> pow += 1↵
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for +=: 'builtin_function_or_method' and 'int'
```

Python se queja de que intentamos sumar un entero y una función. Efectivamente, hay una función *pow* en el módulo *math*. Al importar todo el contenido de *math*, nuestra variable ha sido «machacada» por la función.

Te presentamos algunas de las funciones que encontrarás en el módulo matemático:

sin(*x*)	Seno de *x*, que debe estar expresado en radianes.
cos(*x*)	Coseno de *x*, que debe estar expresado en radianes.
tan(*x*)	Tangente de *x*, que debe estar expresado en radianes.
exp(*x*)	El número *e* elevado a *x*.
ceil(*x*)	Redondeo hacia arriba de *x* (en inglés, «ceiling» significa techo).
floor(*x*)	Redondeo hacia abajo de *x* (en inglés, «floor» significa suelo).
log(*x*)	Logaritmo natural (en base *e*) de *x*.
*log*10(*x*)	Logaritmo decimal (en base 10) de *x*.
sqrt(*x*)	Raíz cuadrada de *x* (del inglés «square root»).

Evitando las coincidencias

Python ofrece un modo de evitar el problema de las coincidencias: indicar solo el nombre del módulo al importar.

```
>>> import math↵
```

De esta forma, todas las funciones del módulo *math* están disponibles, pero usando el nombre del módulo y un punto como prefijo:

```
>>> import math↵
>>> math.sin(0)↵
0.0
>>> math.cos(0)↵
1.0
```

En el módulo matemático se definen, además, algunas constantes de interés:

```
>>> from math import pi, e↵
>>> pi↵
3.141592653589793
>>> e↵
2.718281828459045
```

▶ 30 ¿Qué resultados se obtendrán al evaluar las siguientes expresiones Python? Calcula primero a mano el valor resultante de cada expresión y comprueba, con la ayuda del ordenador, si tu resultado es correcto.

a) *int*(*exp*(2 * *log*(3)))

b) *round*(4**sin*(3 * *pi* / 2))

c) *abs*(*log10*(.01) * *sqrt*(25))

d) *round*(3.21123 * *log10*(1000), 3)

Precisión de los flotantes

Hemos dicho que los argumentos de las funciones trigonométricas deben expresarse en radianes. Como sabrás, $\text{sen}(\pi) = 0$. Veamos qué opina Python:

```
>>> from math import sin, pi↵
>>> sin(pi)↵
1.2246467991473532e-16
```

El resultado que proporciona Python no es cero, sino un número muy próximo a cero: 0.00000000000000012246467991473532. ¿Se ha equivocado Python? No exactamente. Ya dijimos antes que los números flotantes tienen una precisión limitada. El número π está definido en el módulo matemático como 3.14159265358979311599796346854418516159057617187 5, cuando en realidad posee un número infinito de decimales. Así pues, no hemos pedido exactamente el cálculo del seno de π, sino el de un número próximo, pero no exactamente igual. Por otra parte, el módulo matemático hace cálculos mediante algoritmos que pueden introducir errores en el resultado. Fíjate en el resultado de esta sencilla operación:

```
>>> 0.1 - 0.3↵
-0.19999999999999998
```

Los resultados con números en coma flotante deben tomarse como meras aproximaciones de los resultados reales.

2.7.2. Otros módulos de interés

Existe un gran número de módulos, cada uno de ellos especializado en un campo de aplicación determinado. Precisamente, una de las razones por las que Python es un lenguaje potente y extremadamente útil es por la gran colección de módulos con que se distribuye. Hay módulos para el diseño de aplicaciones para web, diseño de interfaces de usuario, compresión de datos, criptografía, multimedia, etc. Y constantemente aparecen nuevos módulos: cualquier programador de Python puede crear sus propios módulos, añadiendo así funciones que simplifican la programación en un ámbito cualquiera y poniéndolas a disposición de otros programadores. Nos limitaremos a presentarte ahora unas pocas funciones de un par de módulos interesantes.

Vamos con otro módulo importante: *sys* (sistema), el módulo de «sistema» (*sys* es una abreviatura del inglés «system»). Este módulo contiene funciones que acceden al sistema operativo y constantes dependientes del computador. Una función importante es *exit*, que aborta inmediatamente la ejecución del intérprete (en inglés significa «salir»). La variable *version*, indica con qué versión de Python estamos trabajando:

```
>>> from sys import version↵
>>> version↵
'3.2.3␣(default,␣Feb␣27␣2014,␣21:31:18)␣\n[GCC␣4.6.3]'
```

Y la variable *platform* permite saber sobre qué sistema operativo se está ejecutando el intérprete:

```
>>> from sys import platform↵
>>> platform↵
'linux2'
```

¡Ojo! Con esto no queremos decirte que las variables *version* o *platform* sean importantísimas y que debas aprender de memoria su nombre y cometido, sino que los módulos de Python contienen centenares de funciones y variables útiles para diferentes propósitos. Un buen programador Python sabe manejarse con los módulos. Existe un manual de referencia que describe todos los módulos estándar de Python. Lo encontrarás con la documentación Python bajo el nombre «*Library reference*» (en inglés significa «referencia de biblioteca») y podrás consultarla con un navegador web.

2.8. Métodos

Los datos de ciertos tipos permiten invocar unas funciones especiales: los denominados «*métodos*». Hemos visto que las funciones se invocan así: *función*(*argumento1*, *argumento2*, *argumento3*...). Los métodos son funciones especiales, pues se invocan del siguiente modo: *argumento1*.*método*(*argumento2*, *argumento3*...). Esta sintaxis recalca el hecho de que, para un método, el primer argumento es muy especial. Es como el sujeto de una frase de la que el método es el verbo.

2.8.1. Unos métodos sencillos para manipular cadenas...

Un método permite, por ejemplo, obtener una versión en minúsculas de la cadena sobre la que se invoca:

```
>>> cadena = 'Un␣EJEMPLO␣de␣Cadena'↵
>>> cadena.lower()↵
'un␣ejemplo␣de␣cadena'
>>> 'OTRO␣EJEMPLO'.lower()↵
'otro␣ejemplo'
```

La sintaxis es diferente de la propia de una llamada a función convencional. Lo primero que aparece es el propio objeto sobre el se efectúa la llamada. El nombre del método se separa del objeto con un punto. Los paréntesis abierto y cerrado al final son obligatorios.

Existe otro método, *upper* («uppercase», en inglés, significa «mayúsculas»), que pasa todos los caracteres a mayúsculas.

```
>>> 'Otro␣ejemplo'.upper()↵
'OTRO␣EJEMPLO'
```

Y otro, *title*, que pasa la inicial de cada palabra a mayúsculas. Te preguntarás para qué puede valer esta última función. Imagina que has hecho un programa de recogida de datos que confecciona un censo de personas y que cada individuo introduce personalmente su nombre en el ordenador. Es muy probable que algunos utilicen solo mayúsculas y otros mayúsculas y minúsculas. Si aplicamos *title* a cada uno de los nombres, todos acabarán en un formato único:

```
>>> 'PEDRO␣F.␣MAS'.title()↵
'Pedro␣F.␣Mas'
>>> 'Juan␣CANO'.title()↵
'Juan␣Cano'
```

Algunos métodos aceptan argumentos. El método *replace*, por ejemplo, recibe como argumento dos cadenas: un patrón y un reemplazo. El método busca el patrón en la cadena sobre la que se invoca el método y sustituye todas sus apariciones por la cadena de reemplazo.

```
>>> 'un␣pequeño␣ejemplo'.replace('pequeño', 'gran')↵
'un␣gran␣ejemplo'
>>> una_cadena = 'abc'.replace('b', '-')↵
>>> una_cadena↵
'a-c'
```

Complejos, decimales y fracciones

Python posee un rico conjunto de tipos de datos. Algunos, como los tipos de datos estructurados, se estudiarán con detalle más adelante. Sin embargo, y dado el carácter introductorio de este texto, no estudiaremos con detalle otros tipos de datos básicos: los números complejos, los números en formato decimal y las fracciones.

Un número complejo puro finaliza siempre con la letra jota, que representa el valor $\sqrt{-1}$. Un número complejo con parte real se expresa sumando la parte real a un complejo puro. He aquí ejemplos de números complejos: 4j, 1 + 2j, 2.0 + 3j, 1 - 0.354j. Y, para acabar, un ejemplo de expresiones aritméticas con complejos:

```
>>> (1 + 3j) / (2 + 1j)↵
(1+1j)
>>> (1 + 2j) * (1 - 2j)↵
(5+0j)
```

Los números de tipo decimal dan una solución al problema de la imprecisión de los flotantes. Esta imprecisión es inaceptable cuando manejamos, por ejemplo, dinero. El tipo *Decimal*, que hemos de importar del módulo *decimal*, maneja decimales con precisión:

```
>>> from decimal import Decimal↵
>>> a = Decimal('0.1')↵
>>> b = Decimal('0.3')↵
>>> a - b↵
Decimal('-0.2')
```

Otro modo de abordar el problema de la imprecisión de los flotantes es usar el tipo *Fraction*, que permite manipular números formados por un numerador y un denominador:

```
>>> from fractions import Fraction↵
>>> Fraction(2, 4)↵
Fraction(1, 2)
>>> Fraction(1, 10) - Fraction(3, 10)↵
Fraction(-1, 5)
```

Tanto *Decimal* como *Fraction* son sensiblemente más lentos que los flotantes. ¡No nos podía salir gratis!

2.8.2. ... y uno mucho más complejo: *format*

Aprender a mostrar la información con formato es esencial para que el usuario encuentre atractiva la forma en que ve los resultados. El método *format* de las cadenas nos proporciona una herramienta fundamental, aunque cuesta un poco dominarlo.

Empezamos por aprender a *interpolar* valores en una cadena, es decir, sustituir una marca especial por el valor de una expresión. Fíjate en esta sentencia:

```
>>> 'El número {0} ha sido interpolado.'.format(1.23)
'El número 1.23 ha sido interpolado.'
```

Los caracteres {0} han sido reemplazados por los caracteres 1.23. Ya sabíamos hacer esto de otro modo:

```
>>> 'El número {0} ha sido interpolado.'.replace('{0}', '1.23')
'El número 1.23 ha sido interpolado.'
```

El método *format* presenta algunas ventajas: su uso conduce a una expresión más breve y el número 1.23 se ha podido suministrar como número (cuando *replace* exige que sea una cadena). Pero las ventajas no acaban ahí. Podemos interporlar más de un valor con una sola llamada a *format*:

```
>>> 'Los números {0} y {1} han sido interpolados.'.format(1.23, 9.9999)
'Los números 1.23 y 9.9999 han sido interpolados.'
```

Cada marca de la forma {*n*}, donde *n* es un número entero, se sustituye por un argumento de *format*: la marca {0} se ha sustituido por el primer argumento y la marca {1} por el segundo. En general, la marca {*n*} se sustituye por el argumento $(n+1)$-ésimo. ¿No habría sido más sencillo para el diseñador de Python que {1} hiciese referencia al primer argumento, {2} al segundo y así sucesivamente? Ya te acostumbrarás a un principio básico de Python: *todas las secuencias empiezan en cero*. (Y no solo de Python: también C, Java y la mayoría de los lenguajes de uso común comparten ese principio).

Las marcas pueden disponerse dentro de la cadena en el orden que desees:

```
>>> 'Los números {1} y {0} han sido interpolados.'.format(1.23, 9.9999)
'Los números 9.9999 y 1.23 han sido interpolados.'
```

▶ 31 ¿Cuál será el resultado de evaluar estas expresiones?

```
>>> '{0}'.format(1)
>>> '{0} {1}'.format(1, 2)
>>> '{0}{1}'.format(1, 2)
>>> '{0}, {1}'.format(1, 2)
>>> '{1}, {0}'.format(1, 2)
>>> '{1}, {1}'.format(1, 2)
>>> '{1}, 1'.format(1, 2)
```

Las marcas pueden modificarse para controlar el aspecto de la información. Imaginemos que deseamos mostrar los valores flotantes redondeados con un solo decimal:

```
>>> 'Los números {0:.1f} y {1:.1f} han sido interpolados.'.format(1.23, 9.9999)
'Los números 1.2 y 10.0 han sido interpolados.'
```

¿Complicado? Las marcas, a las que denominaremos en adelante *marcas de formato*, permiten controlar con precisión el modo en el que se muestran los datos. La forma general de una marca de formato es esta:

{*campo*`!`*marca de conversión*`:`*formato*}

El «campo» es el número que identifica el número de argumento que se desea interpolar en la cadena (aunque admite otros valores). Olvidemos por el momento el fragmento «`!`marca de conversión» y centrémonos en la parte «`:`formato». Su forma general es esta:

[[*relleno*]*alineamiento*][*signo*][#][0][*ancho*][`.`*precisión*][*código de tipo*]

Cada elemento entre corchetes es opcional. Es decir, el fragmento «signo» puede aparecer o no. Fíjate en que los corchetes se anidan en uno de los fragmentos. De acuerdo con ese anidamiento, una marca de formato puede empezar por un *relleno* o no hacerlo, pero solo puede tener un *relleno* si aparece también un *alineamiento*. Y ahora veamos algunas posibilidades para cada uno de esos elementos:

- *relleno*: Carácter con el que rellenar los espacios que requiere un alineamiento (por defecto, espacio en blanco).

- *alineamiento*: Carácter `<` para alinear a la izquierda en el espacio disponible; carácter `>` para alinear a la derecha en el espacio disponible (es el valor por defecto); carácter `^` para centrar en el espacio disponible.
- *signo*: Carácter `+` para forzar la aparición de un signo incluso en números positivos; carácter `-` para indicar que el signo solo debe aparecer con números negativos (es el valor por defecto); un espacio en blanco para indicar que los números positivos deben ir precedidos por un espacio en blanco.
- `#`: Si aparece, los enteros que se muestran en binario, octal o hexadecimal irán precedidos por `0b`, `0o` o `0x`.
- `0`: Si aparece, se usa el carácter `0` para sustituir los espacios en blanco.
- *ancho*: Es un número entero que indica cuántos caracteres queremos que ocupe (como mínimo) el valor representado.
- `.`*precisión*: número de decimales con que queremos representar un número flotante.
- *código de tipo*: carácter que indica el tipo de representación que se desea. Es diferente según el tipo de datos del valor. He aquí algunos de sus posibles valores:
  - números enteros:
    - carácter `b`: en binario.
    - carácter `c`: como carácter Unicode.
    - carácter `d`: en base diez (es el valor por defecto).
    - carácter `o`: en octal.
    - carácter `x`: en hexadecimal.
    - carácter `n`: igual que `d`, pero como número adaptado a la cultura local (en español, por ejemplo, el punto decimal se muestra como una coma).
  - números flotantes:
    - carácter `e`: notación exponente.
    - carácter `f`: notación de punto fijo.
    - carácter `g`: notación en formato general, que solo es exponente para números grandes (es el valor por defecto).
    - carácter `n`: igual que `g`, pero como número adaptado a la cultura local.
    - carácter `%`: muestra el número multiplicado por 100, en formato `f` y seguido de un símbolo de porcentaje.

No has de memorizar esta lista de opciones y posibles valores, pero sí saber dónde está y recurrir a ella cuando la necesites. De todos modos, la mejor manera de ver qué hace exactamente cada opción es estudiar ejemplos de uso.

Empecemos con algunos enteros. Imprimamos el número 123 en su formato por defecto en medio de un texto:

```
>>> 'El␣{0}␣formateado.'.format(123)↵
'El␣123␣formateado.'
```

Ahora veamos cómo afectar de modos diferentes al alineamiento, siempre con una anchura de 10 espacios:

```
>>> 'El␣{0:>10}␣formateado.'.format(123)↵
'El␣␣␣␣␣␣␣123␣formateado.'
>>> 'El␣{0:^10}␣formateado.'.format(123)↵
'El␣␣␣␣123␣␣␣␣␣formateado.'
>>> 'El␣{0:<10}␣formateado.'.format(123)↵
'El␣123␣␣␣␣␣␣␣formateado.'
```

El `0` sustituye los espacios en blanco por un `0`:

```
>>> 'El␣{0:010}␣formateado.'.format(123)↵
'El␣0000000123␣formateado.'
```

Veamos el efecto que consigue especificar un carácter de signo:

```
>>> 'El␣{0:+}␣formateado.'.format(123)↵
'El␣+123␣formateado.'
>>> 'El␣{0:-}␣formateado.'.format(123)↵
'El␣123␣formateado.'
>>> 'El␣{0:␣}␣formateado.'.format(123)↵
'El␣␣123␣formateado.'
```

El código de tipo permite mostrar el número en diferentes bases o incluso como carácter Unicode:

```
>>> 'El␣{0:b}␣formateado.'.format(123)↵
'El␣1111011␣formateado.'
>>> 'El␣{0:c}␣formateado.'.format(123)↵
'El␣{␣formateado.'
>>> 'El␣{0:d}␣formateado.'.format(123)↵
'El␣123␣formateado.'
>>> 'El␣{0:o}␣formateado.'.format(123)↵
'El␣173␣formateado.'
>>> 'El␣{0:x}␣formateado.'.format(123)↵
'El␣7b␣formateado.'
```

Con un # se muestra el prefijo que explicita la base cuando no es la decimal:

```
>>> 'El␣{0:#b}␣formateado.'.format(123)↵
'El␣0b1111011␣formateado.'
>>> 'El␣{0:#o}␣formateado.'.format(123)↵
'El␣0o173␣formateado.'
>>> 'El␣{0:#x}␣formateado.'.format(123)↵
'El␣0x7b␣formateado.'
```

Podemos combinar los diferentes elementos y controlar con mucha precisión el formato:

```
>>> 'El␣{0:+#016b}␣formateado.'.format(123)↵
'El␣+0b0000001111011␣formateado.'
```

No nos extenderemos tanto con las posibilidades de formato para números en coma flotante, pero no nos resistimos a poner algunos ejemplos que deberías analizar:

```
>>> 'El␣{0:e}␣formateado.'.format(123.45)↵
'El␣1.234500e+02␣formateado.'
>>> 'El␣{0:g}␣formateado.'.format(123.45)↵
'El␣123.45␣formateado.'
>>> 'El␣{0:+e}␣formateado.'.format(123.45)↵
'El␣+1.234500e+02␣formateado.'
>>> 'El␣{0:10e}␣formateado.'.format(123.45)↵
'El␣1.234500e+02␣formateado.'
>>> 'El␣{0:10.4g}␣formateado.'.format(123.45)↵
'El␣␣␣␣␣␣123.5␣formateado.'
>>> 'El␣{0:10.2g}␣formateado.'.format(123.45)↵
'El␣␣␣␣␣1.2e+02␣formateado.'
>>> 'El␣{0:10.1g}␣formateado.'.format(123.45)↵
'El␣␣␣␣␣␣␣1e+02␣formateado.'
>>> 'El␣{0:10.0g}␣formateado.'.format(123.45)↵
'El␣␣␣␣␣␣␣1e+02␣formateado.'
>>> 'El␣{0:.1%}␣formateado.'.format(123.45)↵
'El␣12345.0%␣formateado.'
```

Acabamos indicando que hay un pequeño problema: ¿Qué ocurre si queremos mostrar las llaves abierta o cerrada como tales en una cadena sometida a interpolación? Python se liará:

```
>>> 'Una␣{cadena}␣{0}'.format(1)↵
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 'cadena'
```

Las llaves que queramos mostrar como tales y que no sean objeto de sustitución al interpolar deben marcarse con dos apariciones seguidas de cada una de ellas:

```
>>> 'Una␣{{cadena}}␣{0}'.format(1)↵
'Una␣{cadena}␣1'
```

Como ves, hemos entrado en un tema lleno de detalles. Afortunadamente no usaremos muchos de los elementos que acabamos de presentar. Y si alguna vez te hicieran falta, siempre podrás consultar los manuales de ayuda.