2.3. Operadores lógicos y de comparación

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 el valor cierto si y sólo si son ciertos sus dos operandos. Esta es su *tabla de verdad*:

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* sólo cuando ambos operandos son *False*s. 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:

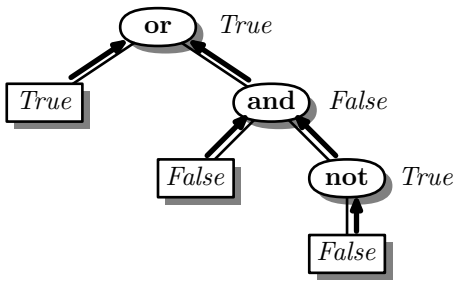
```
>>> True and False ↲
False
>>> not True ↲
False
>>> (True and False) or True ↲
True
>>> True and True or False ↲
True
>>> False and True or True ↲
True
>>> False and True or False ↲
False
```

Has de tener en cuenta la precedencia de los operadores lógicos:

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*:



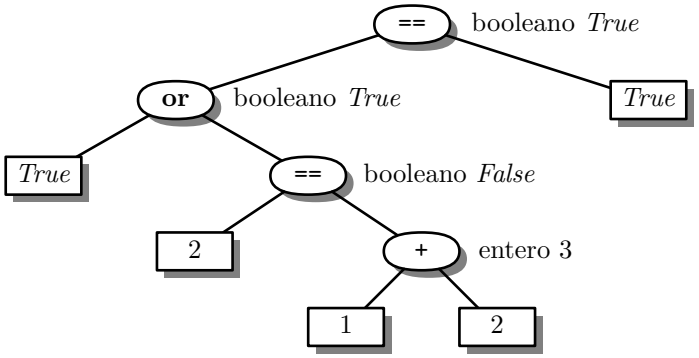
Hay una familia de operadores que devuelven valores booleanos. Entre ellos tenemos a los operadores de comparación, que estudiamos en este apartado. Uno de ellos es el operador de igualdad, que devuelve *True* si los valores comparados son iguales. El operador de igualdad se denota con dos iguales seguidos: **==**. 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 sólo 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[4]:

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

Tabla 2.3: Operadores de comparación.

```
>>> 2 < 1 ↲
False
>>> 1 < 2 ↲
True
>>> 5 > 1 ↲
True
>>> 5 >= 1 ↲
True
>>> 5 > 5 ↲
False
>>> 5 >= 5 ↲
True
>>> 1 != 0 ↲
True
>>> 1 != 1 ↲
```

[4]Hay una forma alternativa de notar la comparación «es distinto de»: también puedes usar el símbolo **<>**. La comparación de desigualdad en el lenguaje de programación C se denota con **!=** y en Pascal con **<>**. Python permite usar cualquiera de los dos símbolos. En este texto sólo usaremos el primero.

```
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
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, resultando 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.

EJERCICIOS

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

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.

```
>>> 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) ↲
```

2.4. 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 ↲
8.1579181568392176
>>> 3.14159265359 * 1.298373 ** 2 ↲
5.2960103355249037
```

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 ↲
8.1579181568392176
>>> pi * r ** 2 ↲
5.2960103355249037
```

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