TIPOS BÁSICOS

En Python los tipos básicos se dividen en:

- Números, como pueden ser `3` (entero), `15.57` (de coma flotante) o `7 + 5j` (complejos)
- Cadenas de texto, como `"Hola Mundo"`
- Valores booleanos: `True` (cierto) y `False` (falso).

Vamos a crear un par de variables a modo de ejemplo. Una de tipo cadena y una de tipo entero:

```
# esto es una cadena
c = "Hola Mundo"

# y esto es un entero
e = 23

# podemos comprobarlo con la función type
type(c)
type(e)
```

Como veis en Python, a diferencia de muchos otros lenguajes, no se declara el tipo de la variable al crearla. En Java, por ejemplo, escribiríamos:

```
String c = "Hola Mundo";
int e = 23;
```

Este pequeño ejemplo también nos ha servido para presentar los comentarios inline en Python: cadenas de texto que comienzan con el carácter # y que Python ignora totalmente. Hay más tipos de comentarios, de los que hablaremos más adelante.

Números

Como decíamos, en Python se pueden representar números enteros, reales y complejos.

Enteros

Los números enteros son aquellos números positivos o negativos que no tienen decimales (además del cero). En Python se pueden representar mediante el tipo `int` (de integer, entero) o el tipo `long` (largo). La única diferencia es que el tipo `long` permite almacenar números más grandes. Es aconsejable no utilizar el tipo `long` a menos que sea necesario, para no malgastar memoria.

El tipo `int` de Python se implementa a bajo nivel mediante un tipo `long` de C. Y dado que Python utiliza C por debajo, como C, y a diferencia de Java, el rango de los valores que puede representar depende de la plataforma.

En la mayor parte de las máquinas el `long` de C se almacena utilizando 32 bits, es decir, mediante el uso de una variable de tipo `int` de Python podemos almacenar números de -2^{31} a $2^{31} - 1$, o lo que es lo mismo, de -2.147.483.648 a 2.147.483.647. En plataformas de 64 bits, el rango es de -9.223.372.036.854.775.808 hasta 9.223.372.036.854.775.807.

El tipo `long` de Python permite almacenar números de cualquier precisión, estando limitados solo por la memoria disponible en la máquina.

Al asignar un número a una variable esta pasará a tener tipo `int`, a menos que el número sea tan grande como para requerir el uso del tipo `long`.

```
# type(entero) devolvería int
entero = 23
```

También podemos indicar a Python que un número se almacene usando `long` añadiendo una L al final:

```
# type(entero) devolvería long
entero = 23L
```

El literal que se asigna a la variable también se puede expresar como un octal, anteponiendo un cero:

```
# 027 octal = 23 en base 10
entero = 027
```

o bien en hexadecimal, anteponiendo un 0x:

```
# 0x17 hexadecimal = 23 en base 10
entero = 0x17
```

Reales

Los números reales son los que tienen decimales. En Python se expresan mediante el tipo `float`. En otros lenguajes de programación, como C, tenemos también el tipo `double`, similar a `float` pero de mayor precisión (double = doble precisión). Python, sin embargo, implementa su tipo `float` a bajo nivel mediante una variable de tipo `double` de C, es decir, utilizando 64 bits, luego en Python siempre se utiliza doble precisión, y en concreto se sigue el estándar IEEE 754: 1 bit para el signo, 11 para el exponente, y 52 para la mantisa. Esto significa que los valores que podemos representar van desde ±2,2250738585072020 x 10^{-308} hasta ±1,7976931348623157×10^{308}.

La mayor parte de los lenguajes de programación siguen el mismo esquema para la representación interna. Pero como muchos sabréis esta tiene sus limitaciones, impuestas por el hardware. Por eso desde Python 2.4 contamos también con un nuevo tipo `Decimal`, para el caso de que se necesite representar fracciones de forma más precisa. Sin embargo este tipo está fuera del alcance de este tutorial, y sólo es necesario para el ámbito de la programación científica y otros relacionados. Para aplicaciones normales podeis utilizar el tipo `float` sin miedo, como ha venido haciéndose desde hace años, aunque teniendo en cuenta que los números en coma flotante no son precisos (ni en este ni en otros lenguajes de programación).

Para representar un número real en Python se escribe primero la parte entera, seguido de un punto y por último la parte decimal.

```
real = 0.2703
```

También se puede utilizar notación científica, y añadir una e (de exponente) para indicar un exponente en base 10. Por ejemplo:

```
real = 0.1e-3
```

sería equivalente a 0.1 x 10-3 = 0.1 x 0.001 = 0.0001

Complejos

Los números complejos son aquellos que tienen parte imaginaria. Si no conocías de su existencia, es más que probable que nunca lo vayas a necesitar, por lo que puedes saltarte este apartado tranquilamente. De hecho la mayor parte de lenguajes de programación carecen de este tipo, aunque sea muy utilizado por ingenieros y científicos en general.

En el caso de que necesitéis utilizar números complejos, o simplemente tengáis curiosidad, os diré que este tipo, llamado `complex` en Python, también se almacena usando coma flotante, debido a que estos números son una extensión de los números reales. En concreto se almacena en una estructura de C, compuesta por dos variables de tipo `double`, sirviendo una de ellas para almacenar la parte real y la otra para la parte imaginaria.

Los números complejos en Python se representan de la siguiente forma:

```
complejo = 2.1 + 7.8j
```

Operadores

Veamos ahora qué podemos hacer con nuestros números usando los operadores por defecto. Para operaciones más complejas podemos recurrir al módulo `math`.

Operadores aritméticos

Operador	Descripción	Ejemplo
+	Suma	`r = 3 + 2  # r es 5`
-	Resta	`r = 4 - 7  # r es -3`

Operador	Descripción	Ejemplo
`-`	Negación	`r = -7 # r es -7`
`*`	Multiplicación	`r = 2 * 6 # r es 12`
`**`	Exponente	`r = 2 ** 6 # r es 64`
`/`	División	`r = 3.5 / 2 # r es 1.75`
`//`	División entera	`r = 3.5 // 2 # r es 1.0`
`%`	Módulo	`r = 7 % 2 # r es 1`

Puede que tengáis dudas sobre cómo funciona el operador de módulo, y cuál es la diferencia entre división y división entera.

El operador de módulo no hace otra cosa que devolvernos el resto de la división entre los dos operandos. En el ejemplo, 7/2 sería 3, con 1 de resto, luego el módulo es 1.

La diferencia entre división y división entera no es otra que la que indica su nombre. En la división el resultado que se devuelve es un número real, mientras que en la división entera el resultado que se devuelve es solo la parte entera.

No obstante hay que tener en cuenta que si utilizamos dos operandos enteros, Python determinará que queremos que la variable resultado también sea un entero, por lo que el resultado de, por ejemplo, `3 / 2` y `3 // 2` sería el mismo: 1.

Si quisiéramos obtener los decimales necesitaríamos que al menos uno de los operandos fuera un número real, bien indicando los decimales

```
r = 3.0 / 2
```

o bien utilizando la función `float` (no es necesario que sepais lo que significa el término función, ni que recordeis esta forma, lo veremos un poco más adelante):

```
r = float(3) / 2
```

Esto es así porque cuando se mezclan tipos de números, Python con-

vierte todos los operandos al tipo más complejo de entre los tipos de los operandos.

Operadores a nivel de bit

Si no conocéis estos operadores es poco probable que vayáis a necesitarlos, por lo que podéis obviar esta parte. Si aún así tenéis curiosidad os diré que estos son operadores que actúan sobre las representaciones en binario de los operandos.

Por ejemplo, si veis una operación como `3 & 2`, lo que estais viendo es un and bit a bit entre los números binarios 11 y 10 (las representaciones en binario de 3 y 2).

El operador *and* (`&`), del inglés "y", devuelve 1 si el primer bit operando es 1 **y** el segundo bit operando es 1. Se devuelve 0 en caso contrario.

El resultado de aplicar and bit a bit a 11 y 10 sería entonces el número binario 10, o lo que es lo mismo, 2 en decimal (el primer dígito es 1 para ambas cifras, mientras que el segundo es 1 sólo para una de ellas).

El operador *or* (`|`), del inglés "o", devuelve 1 si el primer operando es 1 **o** el segundo operando es 1. Para el resto de casos se devuelve 0.

El operador *xor* u or exclusivo (`^`) devuelve 1 si uno de los operandos es 1 y el otro no lo es.

El operador *not* (`~`), del inglés "no", sirve para negar uno a uno cada bit; es decir, si el operando es 0, cambia a 1 y si es 1, cambia a 0.

Por último los operadores de desplazamiento (`<<` y `>>`) sirven para desplazar los bits n posiciones hacia la izquierda o la derecha.

Operador	Descripción	Ejemplo
`&`	and	`r = 3 & 2  # r es 2`
`|`	or	`r = 3 | 2  # r es 3`
`^`	xor	`r = 3 ^ 2  # r es 1`
`~`	not	`r = ~3  # r es -4`

<<	Desplazamiento izq.	`r = 3 << 1 # r es 6`
>>	Desplazamiento der.	`r = 3 >> 1 # r es 1`

Cadenas

Las cadenas no son más que texto encerrado entre comillas simples (`'cadena'`) o dobles (`"cadena"`). Dentro de las comillas se pueden añadir caracteres especiales escapándolos con `\`, como `\n`, el carácter de nueva línea, o `\t`, el de tabulación.

Una cadena puede estar precedida por el carácter `u` o el carácter `r`, los cuales indican, respectivamente, que se trata de una cadena que utiliza codificación Unicode y una cadena *raw* (del inglés, cruda). Las cadenas raw se distinguen de las normales en que los caracteres escapados mediante la barra invertida (`\`) no se sustituyen por sus contrapartidas. Esto es especialmente útil, por ejemplo, para las expresiones regulares, como veremos en el capítulo correspondiente.

```
unicode = u"äóè"
raw = r"\n"
```

También es posible encerrar una cadena entre triples comillas (simples o dobles). De esta forma podremos escribir el texto en varias líneas, y al imprimir la cadena, se respetarán los saltos de línea que introdujimos sin tener que recurrir al carácter `\n`, así como las comillas sin tener que escaparlas.

```
triple = """primera linea
            esto se vera en otra linea"""
```

Las cadenas también admiten operadores como `+`, que funciona realizando una concatenación de las cadenas utilizadas como operandos y `*`, en la que se repite la cadena tantas veces como lo indique el número utilizado como segundo operando.

```
a = "uno"
b = "dos"

c = a + b # c es "unodos"
c = a * 3 # c es "unounouno"
```

Booleanos

Como decíamos al comienzo del capítulo una variable de tipo booleano sólo puede tener dos valores: `True` (cierto) y `False` (falso). Estos valores son especialmente importantes para las expresiones condicionales y los bucles, como veremos más adelante.

En realidad el tipo `bool` (el tipo de los booleanos) es una subclase del tipo `int`. Puede que esto no tenga mucho sentido para tí si no conoces los términos de la orientación a objetos, que veremos más adelante, aunque tampoco es nada importante.

Estos son los distintos tipos de operadores con los que podemos trabajar con valores booleanos, los llamados operadores lógicos o condicionales:

Operador	Descripción	Ejemplo
`and`	¿se cumple a y b?	`r = True and False # r es False`
`or`	¿se cumple a o b?	`r = True or False  # r es True`
`not`	No a	`r = not True       # r es False`

Los valores booleanos son además el resultado de expresiones que utilizan operadores relacionales (comparaciones entre valores):

Operador	Descripción	Ejemplo
`==`	¿son iguales a y b?	`r = 5 == 3 # r es False`
`!=`	¿son distintos a y b?	`r = 5 != 3 # r es True`
`<`	¿es a menor que b?	`r = 5 < 3  # r es False`
`>`	¿es a mayor que b?	`r = 5 > 3  # r es True`

<=	¿es a menor o igual que b?	`r = 5 <= 5 # r es True`
>=	¿es a mayor o igual que b?	`r = 5 >= 3 # r es True`