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

¡Más operadores!

Sólo 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 sólo 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	-6	~00000101	11111010
5 << 1	10	00000101 << 00000001	00001010
5 << 2	20	00000101 << 00000010	00010100
5 << 3	40	00000101 << 00000011	00101000
5 >> 1	2	00000101 >> 00000010	00000010

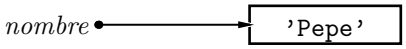
¡Y estos operadores presentan, además, una forma compacta con asignación: **<<=**, **|=**, etc.!

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...

espacios en blanco se muestran así en este texto: «␣». Lo hacemos para que resulte fácil contar los espacios en blanco cuando hay más de uno seguido. Esta cadena, por ejemplo, está formada por tres espacios en blanco: `'␣␣␣'`.

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'
```

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

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 "<stdin>", line 1, in ?
NameError: name 'Pepe' is not defined
```

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

```
>>> 'x' = 2 ↲
SyntaxError: can't assign to literal
```

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

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 (innermost last):
  File "<stdin>", line 1, in ?
TypeError: illegal argument type for built-in operation
```

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 * '-' ↲
'------------------------------------------------------------'
```

EJERCICIOS

▶ 23 Evalúa estas expresiones y sentencias en el orden indicado:

a) *a* = `'b'`

b) *a* + `'b'`

c) *a* + `'a'`

d) *a* * 2 + `'b'` * 3

e) 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) `'a'` * 3 + `'/*'` * 5 + 2 * `'abc'` + `'+'`

b) *palindromo* = `'abcba'`
(4 * `'<'` + *palindromo* + `'>'` * 4) * 2

c) *subcadena* = `'='` + `'-'` * 3 + `'='`
`'10'` * 5 + 4 * *subcadena*

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

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 muy natural: el orden alfabético. En principio, una cadena es menor que otra si la precede al disponerlas en un diccionario. Por ejemplo, `'abajo'` es menor que `'arriba'`.

¿Y cómo se comparan cadenas con caracteres no alfabéticos? Es decir, ¿es `'@@'` menor o mayor que `'abc'`? Python utiliza los códigos ASCII de los caracteres para decidir su orden alfabético (ver tablas en apéndice A). 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.

Otros tipos de datos

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 dos tipos básicos: los números enteros «largos» y los números complejos. Nos limitaremos a presentarlos sucintamente.

El rango de los números flotantes puede resultar insuficiente para ciertas aplicaciones. Python ofrece la posibilidad de trabajar con números con un número de cifras arbitrariamente largo: los enteros «largos». Un entero largo siempre acaba con la letra L. He aquí algunos ejemplos de enteros largos: 1L, -52L, 123764527281763534157182837 4645L. Los números enteros promocionan automáticamente a enteros largos cuando es necesario.

```
>>> 2**30 ↲
1073741824
>>> 2**31 ↲
2147483648L
```

Observa la «L» que aparece al final del segundo resultado: aunque 2 y 31 son números enteros «normales», el resultado de evaluar 2**31 es un entero largo. Esto es así porque los enteros normales se codifican en complemento a 2 de 32 bits, y 2**31 no puede representarse en complemento a 2 de 32 bits.

Si bien los enteros largos resultan cómodos por no producir nunca errores de desbordamiento, debes tener presente que son muy ineficientes: ocupan (mucha) más memoria que los enteros normales y operar con ellos resulta (mucho) más lento.

Finalmente, Python también ofrece la posibilidad de trabajar con números complejos. Un número complejo puro finaliza siempre con la letra *j*, 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: $4j$, $1 + 2j$, $2.0 + 3j$, $1 - 0.354j$.

```
>>> ord('a') ↲
97
```

La función inversa (la que pasa un número a su carácter equivalente) es *chr*.

```
>>> chr(97) ↲
'a'
```

La tabla ASCII presenta un problema cuando queremos ordenar palabras: las letras mayúsculas tienen un valor numérico inferior a las letras minúsculas (por lo que `'Zapata'` precede a `'ajo'`) y las letras acentuadas son siempre «mayores» que sus equivalentes sin acentuar ('abanico' es menor que 'ábaco'). Hay formas de configurar el sistema operativo para que tenga en cuenta los criterios de ordenación propios de cada lengua al efectuar comparaciones, pero esa es otra historia. Si quieres saber más, lee el cuadro titulado «Código ASCII y código IsoLatin-1» y consulta el apéndice A.

EJERCICIOS

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

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

Código ASCII y código IsoLatin-1

En los primeros días de los computadores, los caracteres se codificaban usando 6 o 7 bits. Cada ordenador usaba una codificación de los caracteres diferente, por lo que había problemas de *compatibilidad*: no era fácil transportar datos de un ordenador a otro. Los estadounidenses definieron una codificación estándar de 7 bits que asignaba un carácter a cada número entre 0 y 127: la tabla ASCII (de American Standard Code for Information Interchange). Esta tabla (que puedes consultar en un apéndice) sólo contenía los caracteres de uso común en la lengua inglesa. La tabla ASCII fue enriquecida posteriormente definiendo un código de 8 bits para las lenguas de Europa occidental: la tabla IsoLatin-1, también conocida como ISO-8859-1 (hay otras tablas para otras lenguas). Esta tabla coincide con la tabla ASCII en sus primeros 128 caracteres y añade todos los símbolos de uso común en las lenguas de Europa occidental. Una variante estandarizada es la tabla ISO-8859-15, que es la ISO-8859-1 enriquecida con el símbolo del euro.

(Nota: el código ASCII del carácter `'␣'` es 32, y el del carácter `'a'` es 97.)