ESTRUCTURAS Y TIPOS DE DATOS BÁSICOS

INTRODUCCIÓN

Realizada una primera toma de contacto con Python en el capítulo anterior, dedicaremos el presente a descubrir cuáles son las estructuras de datos y tipos básicos con los que cuenta este lenguaje.

Comenzaremos describiendo y explicando una serie de conceptos básicos propios de este lenguaje y que serán empleados a lo largo del libro. Posteriormente, pasaremos a centrarnos en los tipos básicos, como son, los números y las cadenas de texto. Después, llegará el turno de las estructuras de datos como las tuplas, las listas, los conjuntos y los diccionarios.

CONCEPTOS BÁSICOS

Uno de los conceptos básicos y principales en Python es el *objeto*. Básicamente, podemos definirlo como un componente que se aloja en memoria y que tiene asociados una serie de valores y operaciones que pueden ser realizadas con él. En realidad, los datos que manejamos en el lenguaje cobran vida gracias a estos *objetos*. De momento, estamos hablando desde un punto de vista bastante general; es decir, no debemos asociar este *objeto* al concepto del mismo nombre que se emplea en programación orientada a objetos (OOP). De hecho, un *objeto* en Python puede ser una cadena de texto, un número real, un diccionario o un objeto propiamente dicho, según el paradigma OOP, creado a partir de una clase determinada. En otros lenguajes de programación se emplea el término *estructura de datos* para referirse al *objeto*. Podemos considerar ambos términos como equivalentes.

Habitualmente, un programa en Python puede contener varios componentes. El lenguaje nos ofrece cinco tipos de estos componentes claramente diferenciados. El primero de ellos es el *objeto*, tal y como lo hemos definido previamente. Por otro lado tenemos las *expresiones*, entendidas como una combinación de valores, constantes, variables, operadores y funciones que son aplicadas siguiendo una serle de reglas. Estas expresiones se suelen agrupar formando *sentencias*, consideradas estas como las unidades mínimas ejecutables de un programa. Por último, tenemos los *módulos* que nos ayudan a formar grupos de diferentes sentencias.

Para facilitarnos la programación, Python cuenta con una serie de *objetos* integrados *(built-in)*. Entre las ventajas que nos ofrecen, caben destacar, el ahorro de tiempo, al no ser necesario construir estas estructuras de datos de forma manual, la facilidad para crear complejas estructuras basadas en ellos y el alto rendimiento y mínimo consumo de memoria en tiempo de ejecución. En concreto, contamos con números, cadenas de texto, *booleanos*, listas, diccionarios, tuplas, conjuntos y ficheros. Además, contamos con un tipo de objeto especial llamado *None* que se emplea para asignar un valor nulo. A lo largo de este capítulo describiremos cada uno de ellos, a excepción de los ficheros, de los que se ocupa el capítulo 7.

Antes de comenzar a descubrir los objetos *built-in* de Python, es conveniente explicar qué es y cómo funciona el *tipado dinámico,* del que nos ocuparemos en el siguiente apartado.

Tipado dinámico

Los programadores de lenguajes como Java y C++ están acostumbrados a definir cada variable de un *tipo* determinado. Igualmente ocurre con los objetos, que deben estar asociados a una clase determinada cuando son creados. Sin embargo, Python no trabaja de la misma forma, ya que, al declarar una variable, no se puede indicar su tipo. En tiempo de ejecución, el tipo será asignado a la variable, empleando una técnica conocida como *tipado dinámico.*

¿Cómo es esto posible, cómo diferencia el intérprete entre diferentes tipos y estructuras de datos? La respuesta a estas preguntas hay que buscarla en el funcionamiento interno que el intérprete realiza de la memoria. Cuando se asigna a una variable un valor, el intérprete, en tiempo de ejecución, realiza un proceso que consiste en varios pasos. En primer lugar se crea un objeto en memoria que representará el valor asignado. Seguidamente se comprueba si existe la variable, si no es así se crea una referencia que enlaza la nueva variable con el objeto. Si por el contrario ya existe la variable, entonces, se cambia la referencia hacia el objeto creado. Tanto las variables, como los objetos, se almacenan en diferentes zonas de memoria.

A bajo nivel, las variables se guardan en una tabla de sistema donde se indica a qué objeto referencia cada una de ellas. Los objetos son trozos de memoria con el suficiente espacio para albergar el valor que representan. Por último, las referencias son punteros que enlazan objetos con variables. De esta forma, una variable referencia a un objeto en un determinado momento de tiempo. ¿Cuál es la consecuencia directa de este hecho? Es sencillo, en Python los tipos están asociados a objetos y no a variables. Lógicamente, los objetos conocen de qué tipo son, pero no las variables. Esta es la forma con la que Python consigue implementar el tipado dinámico.

Internamente, el intérprete de Python utiliza un contador de las referencias que se van asignando entre objetos y variables. En función de un algoritmo determinado, cuando estás van cambiando y ya no son necesarias, el recolector

de basura se encargará de marcar como disponible el espacio de memoria ocupado por un objeto que ha dejado de ser referenciado.

Gracias al tipado dinámico podemos, en el mismo bloque de código, asignar diferentes tipos de datos a la misma variable, siendo el intérprete en tiempo de ejecución el que se encargará de crear los objetos y referencias que sean necesarios.

Para clarificar el proceso previamente explicado, nos ayudaremos de un ejemplo práctico. En primer lugar asignaremos el valor numérico *8* a la variable *x* y seguidamente asignaremos a una nueva variable *y* el valor de *x:*

```
>>> x = 8
>>> y = x
```

Después de la ejecución de las sentencias anteriores, en memoria tendríamos una situación como la que muestra la figura 2-1.

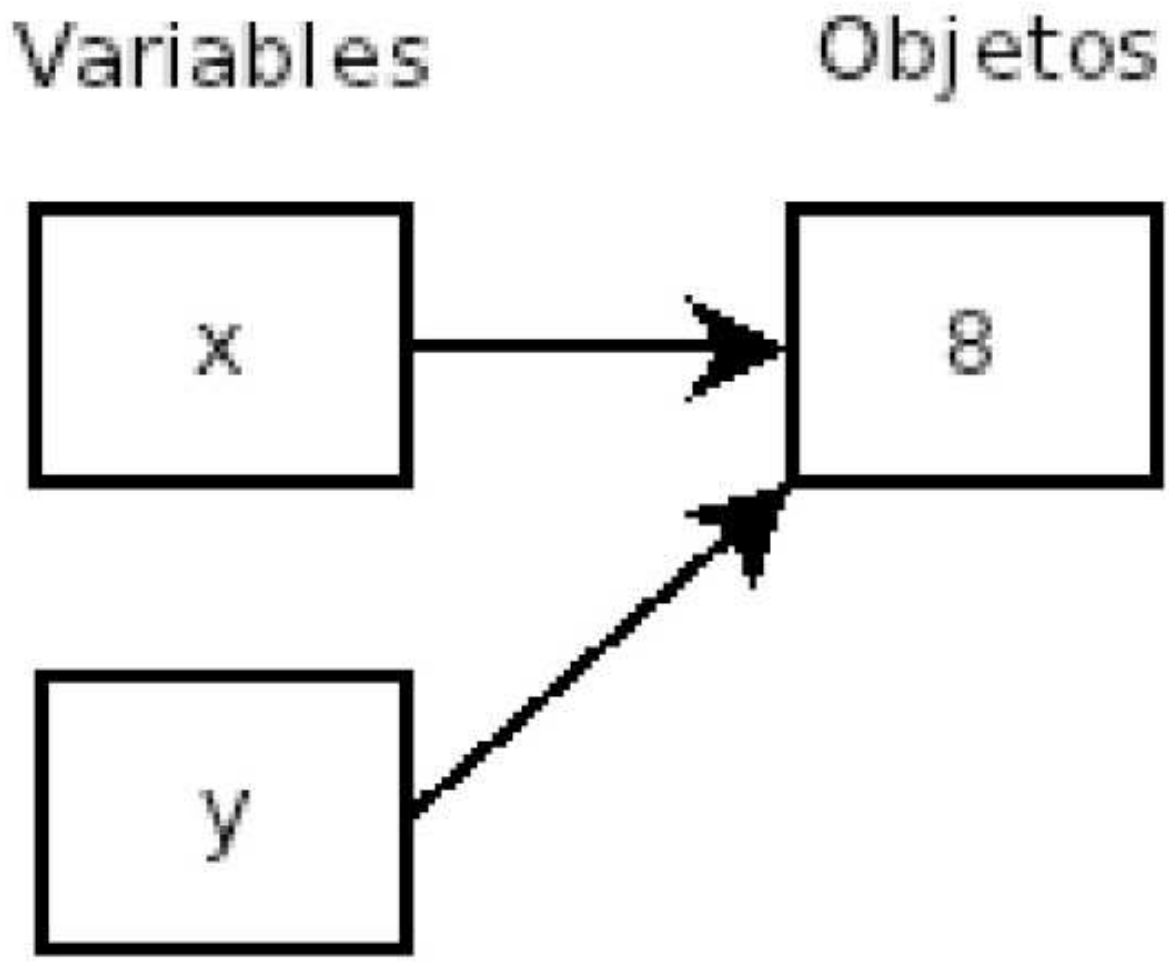

Fig. 2-1 Variables y valor asignado

Posteriormente ejecutamos una nueva sentencia como la siguiente:

```
>>> x = "test"
```

Los cambios efectuados en memoria pueden apreciarse en la figura 2-2, donde comprobaremos cómo ahora las variables tienen distinto valor puesto que apuntan a diferentes objetos.

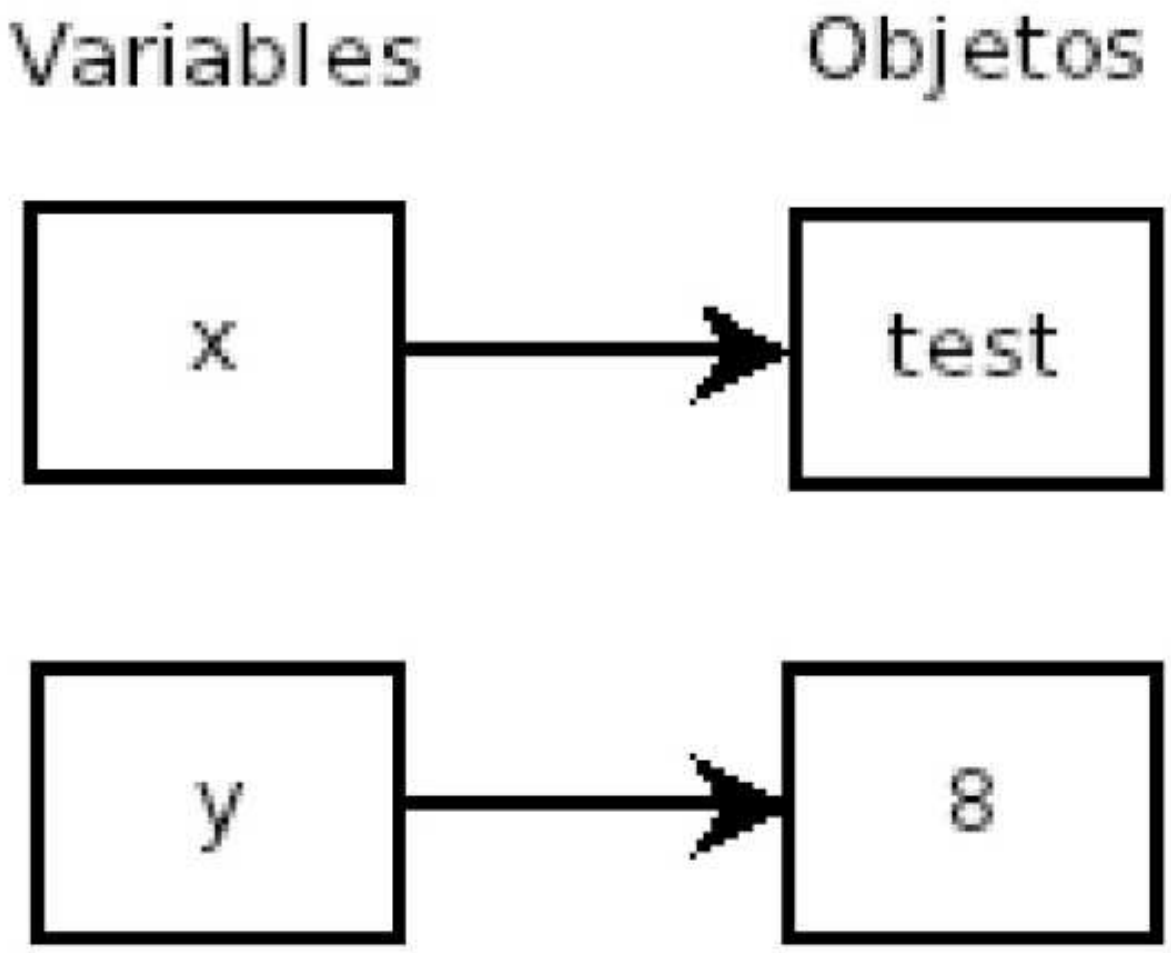

Fig. 2-1 Cambio de valores

Sin embargo, para algunos tipos de objetos, Python realiza la asignación entre objetos y variables de forma diferente a la que hemos explicado previamente. Un ejemplo de este caso es cuando se cambia el valor de un elemento dentro de una lista. Supongamos que definimos una *lista* (un simple *array o* vector) con una serie de valores predeterminados y después, asignamos esta nueva variable a otra diferente llamada *lista_2:*

```
>>> lista_1 = [9, 8, 7]
>>> lista_2 = lista_1
```

Ahora modificamos el segundo elemento de la primera lista, ejecutando la siguiente sentencia:

```
>>> lista_1[2] = 5
```

Como resultado, ambas listas habrán sido modificadas y su valor será el mismo. ¿Por qué se da esta situación? Simplemente, porque no hemos cambiado el objeto, sino un componente del mismo. De esta forma, Python realiza el cambio sobre la marcha, sin necesidad de crear un nuevo objeto y asignar las correspondientes referencias entre variables. Como consecuencia de ello, se ahorra tiempo de procesamiento y memoria cuando el programa es ejecutado por el intérprete.

Una función que nos puede resultar muy útil para ver de qué tipo es una variable es *type()*. Como argumento recibe el nombre de la variable en cuestión y devuelve el tipo precedido de la palabra clave *class*. Gracias a esta función y debido a que una variable puede tomar distintos tipos durante su ejecución, podremos saber a qué tipo pertenece en cada momento de la ejecución del código. Veamos un sencillo ejemplo, a través de las siguientes sentencias y el resultado devuelto por el intérprete:

```
>>> z = 35
>>> type(z)
<class 'int'>
>>> z = "ahora es una cadena de texto"
<class 'str'>
```

NÚMEROS

Como en cualquier lenguaje de programación, en Python, la representación y el manejo de números se hacen prácticamente imprescindibles. Para trabajar con ellos, Python cuenta con una serie de tipos y operaciones integradas, de ambos nos ocuparemos en el presente apartado.

Enteros, reales y complejos

Respecto a los tipos de números soportados por Python, contamos con que es posible trabajar con números enteros, reales y complejos. Además, estos pueden ser representados en decimal, binario, octal y hexadecimal, tal y como veremos más adelante.

De forma práctica, la asignación de un número entero a una variable se puede hacer a través de una sentencia como esta:

```
>>> num_entero = 8
```

Por supuesto, en el contexto de los números enteros, la siguiente expresión también es válida:

```
>>> num_negativo = -78
```

Por otro lado, un número real se asignaría de la siguiente forma:

```
>>> num_real = 4.5
```

En lo que respecta a los números complejos, aquellos formados por una parte *real* y otra *imaginaria*, la asignación sería la siguiente:

```
>>> num_complejo = 3.2 + 7j
```

Siendo también válida la siguiente expresión:

```
>>> num_complex = 5J + 3
```

Como el lector habrá deducido, en los números complejos, la parte imaginaria aparece representada por la letra *j*, siendo también posible emplear la misma letra en mayúscula.

Python 2.x distingue entre dos tipos de enteros en función del tamaño del valor que representan. Concretamente, tenemos los tipos *int* y *long*. En Python 3 esta situación ha cambiado y ambos han sido integrados en un único tipo *int*.

Los valores para números reales que podemos utilizar en Python 3 tienen un amplio rango, gracias a que el lenguaje emplea para su representación un bit para el signo (positivo o negativo), 11 para el exponente y 52 para la mantisa. Esto también implica que se utiliza la precisión doble. Recordemos que en algunos lenguajes de programación se emplean dos tipos de datos para los reales, que varían en función de la representación de su precisión. Es el caso de C, que cuenta con el tipo *float* y *double*. En Python no existe esta distinción y podemos considerar que los números reales representados equivalen al tipo *double* de C.

Además de expresar un número real tal y como hemos visto previamente, también es posible hacerlo utilizando notación científica. Simplemente necesitamos añadir la letra e, que representa el exponente, seguida del valor para él mismo. Teniendo esto en cuenta, la siguiente expresión sería válida para representar al número $0.5*10^{-7}$:

```
>>> num_real = 0.5e-7
```

Desde un punto de vista escrito los *booleanos* no son propiamente números; sin embargo, estos solo pueden tomar dos valores diferentes: *True* (verdadero) o *False* (falso). Dada esta circunstancia, parece lógico utilizar un tipo entero que necesite menos espacio en memoria que el original, ya que, solo necesitamos dos números: 0 y 1. En realidad, aunque Python cuenta con el tipo integrado *bool*, este no es más que una versión personalizada del tipo *int*.

Si necesitamos trabajar con números reales que tengan una precisión determinada, por ejemplo, dos cifras decimales, podemos utilizar la clase *Decimal*. Esta viene integrada en la librería básica que ofrece el intérprete e incluye una serie de funciones, para, por ejemplo, crear un número real con precisión a través de un variable de tipo *float*

Sistemas de representación

Tal y como hemos adelantado previamente, Python puede representar los números enteros en los sistemas decimal, octal, binarlo y hexadecimal. La representación decimal es la empleada comúnmente y no es necesario indicar nada más que el número en cuestión. Sin embargo, para el resto de sistemas es necesario que el número sea precedido de uno o dos caracteres concretos. Por ejemplo, para representar un número en binario nos basta con anteponer los caracteres *Ob*. De esta forma, el número 7 se representaría en binario de la siguiente forma:

```
>>> num_binario = 0b111
```

Por otro lado, para el sistema octal, necesitamos los caracteres Oo, seguidos del número en cuestión. Así pues, el número entero 8 quedaría representado utilizando la siguiente sentencia:

```
>>> num_octal = 0o10
```

En lo que respecta al sistema hexadeclmal, el carácter necesario, tras el número 0, es la letra x. Un ejemplo sería la representación del número 255 a través de la siguiente sentencia:

```
>>> num_hex = 0xff
```

Operadores

Obviamente, para trabajar con números, no solo necesitamos representarlos a través de diferentes tipos, sino también es importante realizar operaciones con ellos. Python cuenta con diversos operadores para aplicar diferentes operaciones numéricas. Dentro del grupo de las aritméticas, contamos con las básicas suma, división entera y real, multiplicación y resta. En cuanto a las operaciones de bajo nivel y entre bits, existen tanto las operaciones NOT y NOR, como XOR y AND. También contamos con operadores para comprobar la igualdad y desigualdad y para realizar operaciones lógicas como AND y OR.

Como en otros lenguajes de programación, en Python también existe la precedencia de operadores, lo que deberemos tener en cuenta a la hora de escribir expresiones que utilicen varios de ellos. Sin olvidar que los paréntesis pueden ser usados para marcar la preferencia entre unas operaciones y otras

dentro de la misma expresión.

La tabla 2-1 resume los principales operadores y operaciones numéricas a las que hacen referencia, siendo o y *b* dos variables numéricas.

Expresión con operador	Operación
a + b	Suma
a - b	Resta
a * b	Multiplicación
a % b	Resto
a / b	División real
a // b	División entera
a ** b	Potencia
a | b	OR (bit)
a ^ b	XOR (bit)
a & b	AND (bit)
a == b	Igualdad
a != b	Desigualdad
a or b	OR (lógica)
a and b	AND (lógica)
not a	Negación (lógica)

Tabla 2-1. Principales operaciones y operadores numéricos

Funciones matemáticas

A parte de las operaciones numéricas básicas, anteriormente mencionadas, Python nos permite aplicar otras muchas funciones matemáticas. Entre ellas, tenemos algunas como el valor absoluto, la raíz cuadrada, el cálculo del valor máximo y mínimo de una lista o el redondo para números reales. Incluso es posible trabajar con operaciones trigonométricas como el seno, coseno y tangente. La mayoría de estas operaciones se encuentran disponibles a través de un módulo (ver definición en capítulo 3) llamado *math*. Por ejemplo, el valor absoluto del número -47,67 puede ser calculado de la siguiente forma:

```
>>> abs(-47,67)
```

Para algunas operaciones necesitaremos importar el mencionado módulo *math*, sirva como ejemplo el siguiente código para calcular la raíz cuadrada del número 169:

```
>>> import math
>>> math.sqrt(169)
```

Otras interesantes operaciones que podemos hacer con números es el cambio de base. Por ejemplo, para pasar de decimal a binario o de octal a hexadecimal. Para ello, Python cuenta con las funciones *int()*, *hex()*, *oct()* y *bin()*. La siguiente sentencia muestra cómo obtener en hexadecimal el valor del entero 16:

```
>>> hex(16)
'0x10'
```

Si lo que necesitamos es el valor octal, por ejemplo, del número 8, bastará con lanzar la siguiente sentencia:

```
>>> oct(8)
'0o10'
```

Debemos tener en cuenta que las funciones de cambio de base admiten como argumentos cualquier representación numérica admitida por Python. Esto quiere decir, que la siguiente expresión también sería válida:

```
>>> bin(0xfe)
'0b11111110'
```

Conjuntos

Definir y operar con conjuntos matemáticos también es posible en Python. La función para crear un conjunto se llama *set()* y acepta como argumentos una serie de valores pasados entre comas, como si se tratara de una cadena de texto. Por ejemplo, la siguiente línea de código define un conjunto tres números diferentes:

```
>>> conjunto = set ('846')
```

Un conjunto también puede ser definido empleando llaves *({})* y separando los elementos por comas. Así pues, la siguiente definición es análoga a la sentencia anterior:

```
>>> conjunto = {8, 4, 6}
```

Operaciones como unión, intersección, creación de subconjuntos y diferencia están disponibles para conjuntos en Python. Algunas operaciones se pueden hacer directamente a través de operadores o bien, llamando al método en cuestión de cada instancia creada. Un ejemplo de ello es la operación intersección. Creemos un nuevo conjunto, utilicemos el operador & y observemos el resultado:

```
>>> conjunto_2 = set('785')
>>> conjunto & conjunto_2
('8')
```

Si en su lugar ejecutamos la siguiente sentencia, veremos que el resultado es el mismo:

```
>>> conjunto.intersection(conjunto_2)
```

A través de los métodos *add()* y *remove()* podemos añadir y borrar elementos de un conjunto.

Si creamos un conjunto con valores repetidos, estos serán automáticamente eliminados, es decir, no formarán parte del conjunto:

```
>>> duplicados = {2, 3, 6, 7, 6, 8, 2, 1}
>>> duplicados

{1, 3, 2, 7, 6, 8}
```

CADENAS DE TEXTO

No cabe duda de que, a parte de los números, las cadenas de texto (*strings*) son otro de los tipos de datos más utilizados en programación. El intérprete de Python integra este tipo de datos, además de una extensa serie de funciones para interactuar con diferentes cadenas de texto.

En algunos lenguajes de programación, como en C, las cadenas de texto no son un tipo integrado como tal en el lenguaje. Esto implica un poco de trabajo extra a la hora de realizar operaciones como la concatenación. Sin embargo, esto no ocurre en Python, lo que hace mucho más sencillo definir y operar con este tipo de dato.

Básicamente, una cadena de texto o *string* es un conjunto inmutable y ordenado de caracteres. Para su representación y definición se pueden utilizar tanto comillas dobles ("), como simples ('). Por ejemplo, en Python, la siguiente sentencia crearía una nueva variable de tipo *string:*

```
>>> cadena = "esto es una cadena de texto"
```

Si necesitamos declarar un string que contenga más de una línea, podemos hacerlo utilizando comillas triples en lugar de dobles o simples:

```
>>> cad_multiple = """Esta cadena de texto
... tiene más de una línea. En concreto, cuenta

... con tres líneas diferentes"""
```

Tipos

Por defecto, en Python 3, todas las cadenas de texto son *Unicode*. Si hemos trabajado con versiones anteriores del lenguaje deberemos tener en mente este hecho, ya que, por defecto, antes se empleaba ASCII. Así pues, cualquier *string* declarado en Python será automáticamente de tipo *Unicode*.

Otra de las novedades de Python 3 con referencia a las cadenas de texto es el tipo de estas que soporta. En concreto son tres las incluidas en esta versión:

Unicode, *byte* y *bytearray*. El tipo *byte* solo admite caracteres en codificación ASCII y, al igual que los de tipo Unicode, son inmutables. Por otro lado, el tipo *bytearray* es una versión mutable del tipo *byte*.

Para declarar un *string* de tipo *byte*, basta con anteponer la letra *b* antes de las comillas:

```
>>> cad = b"cadena de tipo byte"
>>> type(cad)
<class 'bytes'>
```

La declaración de un tipo *bytearray* debe hacerse utilizando la función integrada que nos ofrece el intérprete. Además, es imprescindible indicar el tipo de codificación que deseamos emplear. El siguiente ejemplo utiliza la codificación de caracteres *latin1* para crear un string de este tipo:

```
>>> lat = bytearray("España", 'latin1')
```

Observemos el siguiente ejemplo y veamos la diferencia al emplear diferentes tipos de codificaciones para el mismo string:

```
>>> print(lat)
bytearray(b'Esp\xfla')
>>> bytearray("España", "utf16")
bytearray(b'\xff\xfeE\x00s\x00p\x00a\x00\xf1\x00a\x00')
```

Para las cadenas de texto declaradas por defecto, internamente, Python emplea el tipo denominado *str*. Podemos comprobarlo sencillamente declarando una cadena de texto y preguntando a la función *type():*

```
>>> cadena = "comprobando el tipo str"
>>> type(cadena)
<class 'str'>
```

Realizar conversión entre los distintos tipos de strings es posible gracias a dos tipos de funciones llamadas *encode()* y *decode()*. La primera de ellas se utiliza para transformar un tipo *str* en un tipo *byte*. Veamos cómo hacerlo en el siguiente ejemplo:

```
>>> cad = "es de tipo str"
>>> cad. encode()
b'es de tipo str'
```

La función *decode()* realiza el paso inverso, es decir, convierte un string *byte*

a otro de tipo *str*. Como ejemplo, ejecutaremos las siguientes sentencias:

```
>>> cad = b"es de tipo byte"
>>> cad.decode()
'es de tipo byte'
```

Como el lector habrá podido deducir, cada una de estas funciones solo se encuentra definida para cada tipo. Esto significa que *decode()* no existe para el tipo *str* y que *encode()* no funciona para el tipo *byte*.

Alternativamente, la función *encode()* admite como parámetro un tipo de codificación específico. Si este tipo es indicado, el intérprete utilizará el número de bytes necesarios para su representación en memoria, en función de cada codificación. Recordemos que, para su representación interna, cada tipo de codificación de caracteres requiere de un determinado número de bytes.

Principales funciones y métodos

Para trabajar con strings Python pone a nuestra disposición una serie de funciones y métodos. Las primeras pueden ser invocadas directamente y reciben como argumento una cadena. Por otro lado, una vez que tenemos declarado el string, podemos invocar a diferentes métodos con los que cuenta este tipo de dato.

Una de las funciones más comunes que podemos utilizar sobre strings es el cálculo del número de caracteres que contiene. El siguiente ejemplo nos muestra cómo hacerlo:

```
>>> cad = "Cadena de texto de ejemplo"
>>> len(cad)
26
```

Otro ejemplo de función que puede ser invocada, sin necesidad de declarar una variable de tipo string, es *print()*. En el capítulo anterior mostramos cómo emplearla para imprimir una cadena de texto por la salida estándar.

Respecto a los métodos con los que cuentan los objetos de tipo string, Python incorpora varios de ellos para poder llevar a cabo funcionalidades básicas relacionadas con cadenas de texto. Entre ellas, contamos con métodos para buscar una subcadena dentro de otra, para reemplazar subcadenas, para borrar

espacios en blanco, para pasar de mayúsculas a minúsculas, y viceversa.

La función *find()* devuelve el índice correspondiente al primer carácter de la cadena original que coincide con el buscado:

```
>>> cad ="xyza"
>>> cad.find("y")
1
```

Si el carácter buscado no existe en la cadena, *find()* devolverá -1.

Para reemplazar una serie de caracteres por otros, contamos con el método *replace()*. En el siguiente ejemplo, sustituiremos la subcadena "Hola" por "Adiós":

```
>>> cad = "Hola Mundo"
>>> cad.replace("Hola", "Adiós")
'Adiós Mundo'
```

Obsérvese que *replace()* no altera el valor de la variable sobre el que se ejecuta. Así pues, en nuestro ejemplo, el valor de la variable *cad* seguirá siendo *"Hola Mundo"*.

Los métodos *strip(), lstrip()* y *rstrip()* nos ayudarán a eliminar todos los espacios en blanco, solo los que aparecen a la izquierda y solo los que se encuentran a la derecha, respectivamente:

```
>>> cad = " cadena con espacios en blanco "
>>> cad.strip()
"cadena con espacios en blanco"
>>> cad.lstrip()
"cadena con espacios en blanco "
>>> cad.rstrip()
" cadena con espacios en blanco"
```

El método *upper()* convierte todos los caracteres de una cadena de texto a mayúsculas, mientras que *lower()* lo hace a minúsculas. Veamos un sencillo ejemplo:

```
>>> cad2 = cad.upper()
>>> print(cad2)
"CADENA CON ESPACIOS EN BLANCO"
>>> print(cad3.lower())
" cadena con espacios en blanco "
```

Relacionados con *upper()* y *lower()* encontramos otro método llamado

capitalize(), el cual solo convierte el primer carácter de un string a mayúsculas:

```
>>> cad = "un ejemplo"
>>> cad.capitalize()
'Un ejemplo'
```

En ocasiones puede ser muy útil dividir una cadena de texto basándonos en un carácter que aparece repetidamente en ella. Esta funcionalidad es la que nos ofrece *split()*. Supongamos que tenemos un cadena con varios valores separadas por *;* y que necesitamos una lista donde cada valor se corresponda con los que aparecen delimitados por el mencionado carácter. El siguiente ejemplo nos muestra cómo hacerlo:

```
>>> cad = "primer valor;segundo;tercer valor"
>>> cad.split(";")
['primer valor', 'segundo', 'tercer valor']
```

join() es un método que devuelve una cadena de texto donde los valores de la cadena original que llama al método aparecen separados por un carácter pasado como argumento:

```
>>> "abc".join(',')
'a,b,c'
```

Operaciones

El operador + nos permite concatenar dos strings, el resultado puede ser almacenado en una nueva variable:

```
>>> cad_concat = "Hola" + " Mundo!"
>>> print(cad_concat)
Hola Mundo!
```

También es posible concatenar una variable de tipo string con una cadena o concatenar directamente dos variables. Sin embargo, también podemos prescindir del operador + para concatenar strings. A veces, la expresión es más fácil de leer si empleamos el método *format()*. Este método admite emplear, dentro de la cadena de texto, los caracteres *{}*, entre los que irá, número o el nombre de una variable. Como argumentos del método pueden pasarse variables

que serán sustituidas, en tiempo de ejecución, por los marcadores *{}*, indicados en la cadena de texto. Por ejemplo, veamos cómo las siguientes expresiones son equivalentes y devuelven el mismo resultado:

```
>>> "Hola " + cad2 + ". Otra " + cad3
>>> "Hola {0}. Otra {1}".format(cad2, cad3)
>>> "Hola {cad2}. Otra {cad3}".format(cad2=cad2,cad3=cad3)
```

La concatenación entre strings y números también es posible, siendo para ello necesario el uso de funciones como *int()* y *str()*. El siguiente ejemplo es un caso sencillo de cómo utilizar la función *str():*

```
>>> num = 3
>>> "Número: " + str(num)
```

Interesante resulta el uso del operador * aplicado a cadenas de texto, ya que nos permite repetir un string *n* veces. Supongamos que deseamos repetir la cadena "Hola Mundo" cuatro veces. Para ello, bastará con lanzar la siguiente sentencia:

```
>>> print("Hola Mundo" * 4)
```

Gracias al operador *in* podemos averiguar si un determinado carácter se encuentra o no en una cadena de texto. Al aplicar el operador, como resultado, obtendremos *True o False*, en función de si el valor se encuentra o no en la cadena. Comprobémoslo en el siguiente ejemplo:

```
>>> cad = "Nueva cadena de texto"
>>> "x" in cad
False
```

Un string es inmutable en Python, pero podemos acceder, a través de índices, a cada carácter que forma parte de la cadena:

```
>>> cad = "Cadenas"
>>> print(cad[2])
d
```

Los comentados índices también nos pueden ayudar a obtener subcadenas de texto basadas en la original. Utilizando la variable *cad* del ejemplo anterior podemos imprimir solo los tres primeros caracteres:

```
>>> print(cad[:3])
```

```
Cad
```

En el ejemplo anterior el operador *:* nos ha ayudado a nuestro propósito. Dado que delante del operador no hemos puesto ningún número, estamos indicando que vamos a utilizar el primer carácter de la cadena. Detrás del operador añadimos el número del índice de la cadena de texto que será utilizado como último valor. Así pues, obtendremos los tres primeros caracteres. Los índices negativos también funcionan, simplemente indican que se empieza contar desde el último carácter. La siguiente sentencia devolverá el valor *a*:

```
>>> cad[-2]
```

A continuación, veamos un ejemplo donde utilizamos un número después del mencionado operador:

```
>>> cad [3:]
'enas'
```

TUPLAS

En Python una *tupla* es una estructura de datos que representa una colección de objetos, pudiendo estos ser de distintos tipos. Internamente, para representar una tupla, Python utiliza un array de objetos que almacena referencias hacia otros objetos.

Para declarar una tupla se utilizan paréntesis, entre los cuales deben separarse por comas los elementos que van a formar parte de ella. En el siguiente ejemplo, crearemos una tupla con tres valores, cada uno de un tipo diferente:

```
>>> t = (1, 'a', 3.5)
```

Los elementos de una tupla son accesibles a través del índice que ocupan en la misma, exactamente igual que en un *array:*

```
>>> t [1]
'a'
```

Debemos tener en cuenta que las tuplas son un tipo de dato inmutable, esto significa que no es posible asignar directamente un valor a través del índice.

A diferencia de otros lenguajes de programación, en Python es posible declarar una tupla añadiendo una coma al final del último elemento:

```
>>> t = (1, 3, 'c', )
```

Dado que una tupla puede almacenar distintos tipos de objetos, es posible *anidar* diferentes tuplas; veamos un sencillo ejemplo de ello:

```
>>> t = (1, ('a', 3), 5.6)
```

Una de las peculiaridades de las tuplas es que es un objeto *iterable*; es decir, con un sencillo bucle *for* podemos recorrer fácilmente todos sus elementos:

```
>>> for ele in t:
...    print(ele)
...
1
('a', 3)
5.6
```

Concatenar dos tuplas es sencillo, se puede hacer directamente a través del operador +. Otros de los operadores que se pueden utilizar es *, que sirve para crear una nueva tupla donde los elementos de la original se repiten *n* veces. Observemos el siguiente ejemplo y el resultado obtenido:

```
>>> (’r', 2) * 3
>>> ('r', 2, 'r', 2, 'r', 2)
```

Los principales métodos que incluyen las tupas son *index()* y *count()* El primero de ellos recibe como parámetro un valor y devuelve el índice de la posición que ocupa en la tupla. Veamos el siguiente ejemplo:

```
>>> t = (1, 3, 7)
>>> t.index(3)
1
```

El método *count()* sirve para obtener el número de ocurrencias de un elemento en una tupla:

```
>>> t = (1, 3, 1, 5, 1,)
>>> t.count(1)
3
```

Sobre las tuplas también podemos usar la función integrada *len()*, que nos devolverá el número de elementos de la misma. Obviamente, deberemos pasar la variable tupla como argumento de la mencionada función.

LISTAS

Básicamente, una *lista* es una colección ordenada de objetos, similar al *array dinámico* empleado en otros lenguajes de programación. Puede contener distintos tipos de objetos, es mutable y Python nos ofrece una serie de funciones y métodos integrados para realizar diferentes tipos de operaciones.

Para definir una lista se utilizan corchetes *([])* entre los cuales pueden aparecer diferentes valores separados por comas. Esto significa que ambas declaraciones son válidas:

```
>>> lista = []
>>> li = [2, 'a' , 4]
```

Al igual que las tuplas, las listas son también *iterables*, así pues, podemos recorrer sus elementos empleando un bucle:

```
>>> for ele in li:
...   print(ele)
...
2
'a'
4
```

A diferencia de las tuplas, los elementos de las listas pueden ser reemplazados accediendo directamente a través del índice que ocupan en la lista. De este modo, para cambiar el segundo elemento de nuestra lista *li*, bastaría como ejecutar la siguiente sentencia:

```
>>> 1i [ 1] = ' b'
```

Obviamente, los valores de las listas pueden ser accedidos utilizando el valor del índice que ocupan en la misma:

```
>>> li [2]
4
```

Podemos comprobar si un determinado valor existe en una lista a través del operado *in*, que devuelve *True* en caso afirmativo y *False* en caso contrario:

```
>>> 'a' in li
True
```

Existen dos funciones integradas que relacionan las listas con las tuplas: *list()* y *tuple()*. La primera toma como argumento una tupla y devuelve una lista. En cambio, *tuple()* devuelve una tupla al recibir como argumento una lista. Por ejemplo, la siguiente sentencia nos devolverá una tupla:

```
>>> tuple(li)
(2, 'a', 4)
```

Operaciones como la suma (+) y la multiplicación (*) también pueden ser aplicadas sobre listas. Su funcionamiento es exactamente igual que en las tuplas.

Inserciones y borrados

Para añadir un nuevo elemento a una lista contamos con el método *append()*. Como parámetro hemos de pasar el valor que deseamos añadir y este será insertado automáticamente al final de la lista. Volviendo a nuestra lista ejemplo de tres elementos, uno nuevo quedaría insertado a través de la siguiente sentencia:

```
>>> li.append('nuevo')
```

Nótese que, para añadir un nuevo elemento, no es posible utilizar un índice superior al número de elementos que contenga la lista. La siguiente sentencia lanza un error:

```
>>> li [4] = 23
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

Sin embargo, el método *insert()* sirve para añadir un nuevo elemento especificando el índice. Si pasamos como índice un valor superior al número de elementos de la lista, el valor en cuestión será insertado al final de la misma, sin tener en cuenta el índice pasado como argumento. De este modo, las siguientes sentencias producirán el mismo resultado, siendo *'c'* el nuevo elemento que será insertado en la lista *li:*

```
>>> li.insert(3, 'c')
>>> li.insert(12, 'c')
```

Por el contrario, podemos insertar un elemento en una posición determinada cuyo índice sea menor al número de valores de la lista. Por ejemplo, para insertar un nuevo elemento en la primera posición de nuestra lista *li*, bastaría con ejecutar la siguiente sentencia:

```
>>> li.insert(0, 'd')
>>> li
>>> ['d’, 2, 'a', 4]
```

Si lo que necesitamos es borrar un elemento de una lista, podemos hacerlo gracias a la función *del(),* que recibe como argumento la lista junto al índice que referencia al elemento que deseamos eliminar. La siguiente sentencia ejemplo borra el valor *2* de nuestra lista *li:*

```
>>> del(li[1])
```

Como consecuencia de la sentencia anterior, la lista queda reducida en un elemento. Para comprobarlo contamos con la función *len(),* que nos devuelve el número de elementos de la lista:

```
>>> len(li)
2
```

Obsérvese que la anterior función también puede recibir como argumento una tupla o un string. En general, *len()* funciona sobre tipos de objetos iterables.

También es posible borrar un elemento de una lista a través de su valor. Para ello contamos con el método *remove():*

```
>>> li.remove('d')
```

Si un elemento aparece repetido en la lista, el método *remove()* solo borrará la primera ocurrencia que encuentre en la misma.

Otro método para eliminar elementos es *pop().* A diferencia de *remove(), pop()* devuelve el elemento borrado y recibe como argumento el índice del elemento que será eliminado. Si no se pasa ningún valor como índice, será el último elemento de la lista el eliminado. Este método puede ser útil cuando necesitamos ambas operaciones (borrar y obtener el valor) en una única sentencia.

Ordenación

Los elementos de una lista pueden ser ordenados a través del método *sort() o* utilizando la función *sorted()*. Como argumento se puede utilizar *reverse* con el valor *True* o *False*. Por defecto, se utiliza el segundo valor, el cual indica que la lista será ordenada de mayor a menor. Si por el contrario el valor es *True*, la lista será ordenada inversamente. Veamos un ejemplo para ordenar una lista de enteros:

```
>>> lista = [3, 1, 9, 8, 7]
>>> sorted(lista)
[1, 3, 7, 8, 9]
>>> sorted(lista, reverse=True)
[9, 8, 7, 3, 1]
>>> lista
[3, 1, 9, 8, 7]
```

Como el lector habrá podido observar, la lista original ha quedado inalterada. Sin embargo, si en lugar de utilizar la función *sorted()*, empleamos el método *sort()*, la lista quedará automáticamente modificada. Ejecutemos las siguientes setencias para comprobarlo:

```
>>> lista.sort()
>>> lista
[1, 3, 7, 8, 9]
```

Tanto para aplicar *sort()* como *sorted()* debemos tener en cuenta que la lista que va a ser ordenada contiene elementos que son del mismo tipo. En caso contrario, el intérprete de Python lanzará un error. No obstante, es posible realizar ordenaciones de listas con elementos de distinto tipo si es el programador el encargado de establecer el criterio de ordenación. Para ello, contamos con el parámetro *key* que puede ser pasado como argumento. El valor del mismo puede ser una función que fijará cómo ordenar los elementos. Además, el mencionado parámetro también puede ser utilizado para cambiar la forma de ordenar que emplee el intérprete por defecto, aunque los elementos sean del mismo tipo. Supongamos que definimos la siguiente lista:

```
>>> lis = ['aA', 'Ab', 'Cc', ’ca']
```

Ahora ordenaremos con la función *sorted()* sin ningún parámetro adicional y

observaremos que el criterio de ordenación que utiliza el intérprete, por defecto, es ordenar primero las letras mayúsculas:

```
>>> sorted(lis)
['Ab', 'Cc' , 'aA', 'ca']
```

Sin embargo, al pasar como argumento un determinado criterio de ordenación, el resultado varía:

```
>>> sorted(lis, key=str.lower)
['aA', 'Ab', 'ca', 'Cc']
```

Otro método que contienen las listas relacionado con la ordenación de valores es *reverse()*, que automáticamente ordena una lista en orden inverso al que se encuentran sus elementos originales. Tomando el valor de la última lista de nuestro ejemplo, llamaremos al método para ver qué ocurre:

```
>>> lista.reverse()

>>> lista

[9, 8, 7, 3, 1]
```

Los métodos y funciones de ordenación no solo funcionan con números, sino también con caracteres y con cadenas de texto:

```
>>> lis = ['be', 'ab', 'cc', 'aa', 'cb']
>>> lis.sort()
>>> lis

['aa', 'ab','be', 'cb', 'cc']
```

Comprensión

La *comprensión* de listas es una construcción sintáctica de Python que nos permite declarar una lista a través de la creación de otra. Esta construcción está basada en el principio matemático de la teoría de comprensión de conjuntos. Básicamente, esta teoría afirma que un conjunto se define por comprensión cuando sus elementos son nombrados a través de sus características. Por ejemplo, definimos el conjunto *S* como aquel que está formado por todos los

meses del año: *S = {meses del año}*

Veamos un ejemplo práctico para utilizar la mencionada construcción sintáctica en Python:

```
>>> lista = [ele for ele in (1, 2, 3)]
```

Como resultado de la anterior sentencia, obtendremos una lista con tres elementos diferentes:

```
>>> print(lista)
[1, 2, 3]
```

Gracias a la comprensión de listas podemos definir y crear listas ahorrando líneas de código y escribiendo el mismo de forma más elegante. Sin la comprensión de listas, deberíamos ejecutar las siguientes sentencias para lograr el mismo resultado:

```
>>> lista = []
>>> for ele in (1, 2, 3):
...   lista.append(ele)
...
```

Matrices

Anidando listas podemos construir *matrices* de elementos. Estas estructuras de datos son muy útiles para operaciones matemáticas. Debemos tener en cuenta que complejos problemas matemáticos son resueltos empleando matrices. Además, también son prácticas para almacenar ciertos datos, aunque no se traten estrictamente de representar matrices en el sentido matemático.

Por ejemplo, una matriz matemática de dos dimensiones puede definirse de la siguiente forma:

```
>>> matriz = [[1, 2, 3],[4, 5, 6]]
```

Para acceder al segundo elemento de la primera matriz, bastaría con ejecutar la siguiente sentencia:

```
>>> matriz = [0][1]
```

Asimismo, podemos cambiar un elemento directamente:

```
>>> matriz[0][1] = 33
>>> m

[[1, 33, 3],[4, 5, 6]]
```

DICCIONARIOS

Un *diccionario* es una estructura de datos que almacena una serie de valores utilizando otros como referencia para su acceso y almacenamiento. Cada elemento de un diccionario es un par *clave-valor* donde el primero debe ser único y será usado para acceder al valor que contiene. A diferencia de las tuplas y las listas, los diccionarios no cuentan con un orden específico, siendo el intérprete de Python el encargado de decidir el orden de almacenamiento. Sin embargo, un diccionario es iterable, mutable y representa una colección de objetos que pueden ser de diferentes tipos.

Gracias a su flexibilidad y rapidez de acceso, los diccionarios son una de las estructuras de datos más utilizadas en Python. Internamente son representadas como una tabla *hash*, lo que garantiza la rapidez de acceso a cada elemento, además de permitir aumentar dinámicamente el número de ellos. Otros muchos lenguajes de programación hacen uso de esta estructura de datos, con la diferencia de que es necesario implementar la misma, así como las operaciones de acceso, modificación, borrado y manejo de memoria. Python ofrece la gran ventaja de incluir los diccionarios como estructuras de datos integradas, lo que facilita en gran medida su utilización.

Para declarar un diccionario en Python se utilizan las llaves *({})* entre las que se encuentran los pares clave-valor separados por comas. La clave de cada elemento aparece separada del correspondiente valor por el carácter *:*. El siguiente ejemplo muestra la declaración de un diccionario con tres valores:

```
>>> diccionario = {'a': 1, 'b': 2, 'c': 3}
```

Alternativamente, podemos hacer uso de la función *dict()* que también nos permite crear un diccionario. De esta forma, la siguiente sentencia es equivalente a la anterior:

```
>>> diccionario = dict(a=1, b=2, c=3)
```

Acceso, inserciones y borrados

Como hemos visto previamente, para acceder a los elementos de las listas y las tuplas, hemos utilizado el índice en función de la posición que ocupa cada elemento. Sin embargo, en los diccionarios necesitamos utilizar la clave para acceder al valor de cada elemento. Volviendo a nuestro ejemplo, para obtener el valor indexado por la clave *'c'* bastará con ejecutar la siguiente sentencia:

```
>>> diccionario 'c']
3
```

Para modificar el valor de un diccionario, basta con acceder a través de su clave:

```
>>> diccionario['b'] = 28
```

Añadir un nuevo elemento es tan sencillo como modificar uno ya existente, ya que si la clave no existe, automáticamente Python la añadirá con su correspondiente valor. Así pues, la siguiente sentencia insertará un nuevo valor en nuestro diccionario ejemplo:

```
>>> diccionario['d'] = 4
```

Tres son los métodos principales que nos permiten iterar sobre un diccionario: *items()*, *values()* y *keys()*. El primero nos da acceso tanto a claves como a valores, el segundo se encarga de devolvernos los valores, y el tercero y último es el que nos devuelve las claves del diccionario. Veamos estos métodos en acción sobre el diccionario original que declaramos previamente:

```
>>> for k, v in diccionario.items():
...   print("clave={0}, valor={1}".format(k, v))
...
clave=a, valor=1
clave=b, valor=2
clave=c, valor=3

>>> for k in diccionario.keys():
...   print("clave={0}".format(k))
...
clave=a
clave=b
clave=c

>>> for v in diccionario.values():
...   print("valor={0}".format(v))
...
```

```
valor=1
valor=2
valor=3
```

Por defecto, si iteramos sobre un diccionario con un bucle *for*, obtendremos las claves del mismo sin necesidad de llamar explícitamente al método *keys()*:

```
>>> for k in diccionario:
...   print(k)
a
b
c
```

A través del método *keys()* y de la función integrada *list()* podemos obtener una lista con todas las claves de un diccionario:

```
>>> list(diccionario.keys())
```

Análogamente es posible usar *values()* junto con la función *list()* para obtener un lista con los valores del diccionario. Por otro lado, la siguiente sentencia nos devolverá una lista de tuplas, donde cada una de ellas contiene dos elementos, la clave y el valor de cada elemento del diccionario:

```
>>> list(diccionario.items())
[ ('a', 1), (’b', 2), ('c', 3)]
```

La función integrada *del()* es la que nos ayudará a eliminar un valor de un diccionario. Para ello, necesitaremos pasar la clave que contiene el valor que deseamos eliminar. Por ejemplo, para eliminar el valor que contiene la clave *'c'* de nuestro diccionario, basta con ejecutar:

```
>>> del(diccionario['b'])
```

El método *pop()* también puede ser utilizado para borrar eliminar elementos de un diccionario. Su funcionamiento es análogo al explicado en el caso de las listas.

Otra función integrada, en este caso *len()*, también funciona sobre los diccionarios, devolviéndonos el número total de elementos contenidos.

El operador *in* en un diccionario sirve para comprobar si una clave existe. En caso afirmativo devolverá el valor *True* y *False* en otro caso:

```
>>> 'x' in diccionario
False
```

Comprensión

De forma similar a las listas, los diccionarios pueden también ser creados por comprensión. El siguiente ejemplo muestra cómo crear un diccionario utilizando la iteración sobre una lista:

```
>>> {k: k+1 for k in (1, 2, 3)}
{1: 2, 3: 4, 4: 5}
```

La comprensión de diccionarios puede ser muy útil para inicializar un diccionario a un determinado valor, tomando como claves los diferentes elementos de una lista. Veamos cómo hacerlo a través del siguiente ejemplo que crea un diccionario inicializándolo con el valor *1* para cada clave:

```
>>> {clave: 1 for clave in ['x', 'y', 'z']}
{'x': 1, 'y': 1, 'z': 1}
```

Ordenación

A diferencia de las listas, los diccionarios no tienen el método *sort()*, pero sí que es posible utilizar la función integrada *sorted()* para obtener una lista ordenada de las claves contenidas. Volviendo a nuestro diccionario ejemplo inicial, ejecutaremos la siguiente sentencia:

```
>>> sorted(diccionario)
['a', 'b', 'c']
```

También podemos utilizar el parámetro *reverse* con el mismo resultado que en las listas:

```
>>> sorted(diccionario, reverse=True)
['c', ' b' , 'a']
```