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:

```
>>> li[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
```

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.

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: *ítems()*, *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
```