Python

Guía para ser un Pythonista

Juan José Lozano Gómez – j2logo.com

10
Tipo dict

Índice de contenidos

Introducción

Continuamos con un tema relacionado con los tipos de datos colección.

En esta ocasión nos vamos a centrar en un tipo muy especial, el tipo diccionario.

Tipo dict

La clase o tipo `dict` es de tipo *Mapa*. Un *mapa* es un objeto contenedor que admite búsquedas arbitrarias de claves e implementa los métodos especificados en las clases abstractas `Mapping` o `MutableMapping`.

En concreto, la clase `dict` es un **array asociativo, mutable y ordenado**, en el que los valores son asociados a claves arbitrarias.

Diría que, junto con el tipo `list`, es uno de los tipos colección más importantes del lenguaje.

A diferencia de los tipos secuenciales (`list`, `tuple`, `range` o `str`, entre otros), que son indexados por un índice numérico, los diccionarios son indexados por claves. Estas claves siempre deben ser de un tipo *inmutable*, concretamente un tipo `hashable`.

NOTA: Un objeto es *hashable* si tiene un valor de hash que no cambia durante todo su ciclo de vida. En principio, los objetos que son instancias de clases definidas por el usuario son *hashables*. También lo son la mayoría de tipos *inmutables* definidos por Python (por ejemplo: *int*, *float*, *str* o *tuple*).

Piensa siempre en un diccionario como un contenedor de pares `clave: valor`, en el que la *clave* puede ser de cualquier tipo *hashable* y es única en el diccionario que la contiene. Generalmente, se suelen usar como claves los tipos `int` y `str` aunque, como te he dicho, cualquier tipo *hashable* puede ser una clave.

Las **principales operaciones** que se suelen realizar con diccionarios son **almacenar un valor asociado a una clave** y **recuperar un valor a partir de una clave**. Esta es la esencia de los diccionarios y es aquí donde son realmente importantes.

En un diccionario, el acceso a un elemento a partir de una clave es una operación realmente rápida, eficaz y que consume pocos recursos si lo comparamos con cómo lo haríamos con otros tipos de datos.

Otras características que resaltar del tipo `dict`:

- Es un tipo mutable, es decir, su contenido se puede modificar después de haber sido creado.
- Es un tipo ordenado. Preserva el orden en que se insertan los pares `clave: valor`.

Cómo crear un diccionario

En Python hay varias formas de crear un diccionario. Te las presento todas a continuación.

La más simple es encerrar una secuencia de pares `clave: valor` separados por comas entre llaves `{}`:

```
>>> d = {1: 'hola', 'a': 'b', 'c': 27}
```

En el diccionario anterior, el entero `1` y las cadenas `'a'` y `'c'` son las *claves*. Como ves, se pueden mezclar claves y valores de distinto tipo sin problema.

Para crear un diccionario vacío, simplemente asigna a una variable el valor `{}`.

También se puede usar el constructor de la clase `dict()` de varias maneras:

- **Sin parámetros.** Esto creará un diccionario vacío.
- Con pares `clave: valor` encerrados entre llaves.
- **Con argumentos con nombre**. El nombre del argumento será la clave en el diccionario. En este caso, las claves solo pueden ser identificadores

válidos y mantienen el orden en el que se indican. No se podría, por ejemplo, tener números enteros como claves.

- **Pasando un iterable.** En este caso, cada elemento del iterable debe ser también un iterable con solo dos elementos. El primero se toma como clave del diccionario y el segundo como valor. Si la clave aparece varias veces, el valor que prevalece es el último.

Veamos un ejemplo con todo lo anterior. Vamos a crear el mismo diccionario de todos los modos que te he explicado:

```
# 1. Pares clave: valor encerrados entre llaves
>>> d = {'uno': 1, 'dos': 2, 'tres': 3}
>>> d
{'uno': 1, 'dos': 2, 'tres': 3}

# 2. Argumentos con nombre
>>> d2 = dict(uno=1, dos=2, tres=3)
>>> d2
{'uno': 1, 'dos': 2, 'tres': 3}

# 3. Pares clave: valor encerrados entre llaves
>>> d3 = dict({'uno': 1, 'dos': 2, 'tres': 3})
>>> d3
{'uno': 1, 'dos': 2, 'tres': 3}
```

```
# 4. Iterable que contiene iterables con
# dos elementos
>>> d4 = dict([('uno', 1), ('dos', 2), ('tres',
3)])
>>> d4
{'uno': 1, 'dos': 2, 'tres': 3}

# 5. Diccionario vacío
>>> d5 = {}
>>> d5
{}

# 6. Diccionario vacío usando el constructor
>>> d6 = dict()
>>> d6
{}
```

En lo que sigue, te mostraré las principales operaciones que se pueden realizar sobre objetos de la clase `dict`.

Cómo acceder a los elementos de un diccionario

Acceder a un elemento de un diccionario es una de las principales operaciones por las que existe este tipo de dato. El acceso a un valor se realiza mediante indexación de la clave. Para ello, simplemente encierra entre corchetes la clave `d[clave]`. En caso de que la clave no exista, se lanzará la excepción `KeyError`.

```
>>> d = {'uno': 1, 'dos': 2, 'tres': 3}
>>> d['dos']
2

>>> d[4]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 4
```

La clase `dict` también ofrece el método `get(clave[, valor por defecto])`. Este método devuelve el valor correspondiente a la clave `clave`. En caso de que la clave no exista no lanza ningún error, sino que devuelve el segundo argumento `valor por defecto`. Si no se proporciona este argumento, se devuelve el valor `None`.

```
>>> d = {'uno': 1, 'dos': 2, 'tres': 3}
>>> d.get('uno')
1
# Devuelve 4 como valor por defecto
>>> d.get('cuatro', 4)
4
# Devuelve None como valor por defecto
>>> a = d.get('cuatro')
>>> a
>>> type(a)
<class 'NoneType'>
```

Bucle for y dict

Debido a su naturaleza, hay varias formas de recorrer los elementos de un diccionario: recorrer solo las claves, solo los valores o recorrer a la vez las claves y los valores.

A continuación, te muestro cómo usar el bucle `for` para recorrer un diccionario de todas las formas posibles:

```
>>> d = {'uno': 1, 'dos': 2, 'tres': 3}

# Recorrer las claves del diccionario
>>> for e in d:
...     print(e)
...
uno
dos
tres

# Recorrer las claves del diccionario
>>> for k in d.keys():
...     print(k)
...
uno
dos
tres
```

```
# Recorrer los valores del diccionario
>>> for v in d.values():
...     print(v)
...
1
2
3

# Recorrer los pares clave valor
>>> for i in d.items():
...     print(i)
...
('uno', 1)
('dos', 2)
('tres', 3)
```

Añadir elementos a un diccionario en Python

Como te decía, la clase `dict` es mutable, por lo que se pueden añadir, modificar y/o eliminar elementos después de haberse creado un objeto de este tipo.

Para añadir un nuevo elemento a un diccionario existente se usa el operador de asignación `=`. A la izquierda del operador aparece el objeto diccionario con la nueva clave entre corchetes `[]` y a la derecha el valor que se asocia a dicha clave.

```
>>> d = {'uno': 1, 'dos': 2}
>>> d
{'uno': 1, 'dos': 2}

# Añade un nuevo elemento al diccionario
>>> d['tres'] = 3
>>> d
{'uno': 1, 'dos': 2, 'tres': 3}
```

NOTA: Si la clave ya existe en el diccionario, se actualiza su valor.

También existe el método `setdefault(clave[, valor])`. Este método devuelve el valor de la clave si ya existe y, en caso contrario, le asigna el valor que se pasa como segundo argumento. Si no se especifica este segundo argumento, por defecto es `None`.

```
>>> d = {'uno': 1, 'dos': 2}
>>> d.setdefault('uno', 1.0)
1
>>> d.setdefault('tres', 3)
3
>>> d.setdefault('cuatro')
>>> d
{'uno': 1, 'dos': 2, 'tres': 3, 'cuatro': None}
```

Actualizar elementos

En el apartado anterior hemos visto que, para actualizar el valor asociado a una clave, simplemente hay que asignar un nuevo valor a dicha clave del diccionario:

```
>>> d = {'uno': 1, 'dos': 2}
>>> d
{'uno': 1, 'dos': 2}
>>> d['uno'] = 1.0
>>> d
{'uno': 1.0, 'dos': 2}
```

Por otro lado, también se puede actualizar un diccionario a través del método `update()`. Este método acepta como argumento un diccionario, un objeto iterable o argumentos con nombre. En caso de que se indique una clave que ya exista, se actualizará su valor:

```
>>> d = {'uno': 1, 'dos': 2}
>>> d.update(tres=3)
>>> d
{'uno': 1, 'dos': 2, 'tres': 3}
>>> d.update({'tres': 5})
>>> d
{'uno': 1, 'dos': 2, 'tres': 5}
```

Eliminar un elemento

En Python existen diversos modos de eliminar un elemento de un diccionario. Son los siguientes:

- `pop(clave [, valor por defecto])`: Si la `clave` está en el diccionario, elimina el elemento y devuelve su valor; si no, devuelve el `valor por defecto`. Si no se proporciona el valor por defecto y la clave no está en el diccionario, se lanza la excepción `KeyError`.
- `popitem()`: Elimina el último par `clave: valor` del diccionario y lo devuelve. Si el diccionario está vacío se lanza la excepción `KeyError`. (**NOTA**: *En versiones anteriores a Python 3.7, se elimina/devuelve un par aleatorio, no se garantiza que sea el último*).
- `del d[clave]`: Elimina el par `clave: valor`. Si no existe la clave, se lanza la excepción `KeyError`.
- `clear()`: Borra todos los pares `clave: valor` del diccionario.

```
>>> d = {'uno': 1, 'dos': 2, 'tres': 3,
'cuatro': 4, 'cinco': 5}

# Elimina un elemento con pop()
>>> d.pop('uno')
1
>>> d
{'dos':  2,  'tres':  3,  'cuatro':  4,
'cinco': 5}

# Trata de eliminar una clave con pop()
# que no existe
>>> d.pop(6)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 6

# Elimina un elemento con popitem()
>>> d.popitem()
('cinco', 5)
>>> d
{'dos': 2, 'tres': 3, 'cuatro': 4}

# Elimina un elemento con del
>>> del d['tres']
>>> d
{'dos': 2, 'cuatro': 4}
```

```
# Trata de eliminar una clave con del
# que no existe
>>> del d['seis']
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 'seis'

# Borra todos los elementos
# del diccionario
>>> d.clear()
>>> d
{}
```

Número de elementos de un diccionario

Al igual que sucede con otros tipos contenedores, se puede usar la función de Python `len()` para obtener el número de elementos o longitud de un diccionario.

```
>>> d = {'uno': 1, 'dos': 2, 'tres': 3}
>>> len(d)
3
```

Comprobar si un elemento está en un diccionario

Al operar con diccionarios, se puede usar el operador de pertenencia `in` para comprobar si una clave está contenida, o no, en un diccionario. Esto resulta útil, por ejemplo, para asegurarnos de que una clave existe antes de intentar eliminarla.

```
>>> d = {'uno': 1, 'dos': 2, 'tres': 3}
>>> print('uno' in d)
True
>>> print(1 in d)
False
>>> print(1 not in d)
True
# Intenta eliminar la clave 1 si existe
>>> if 1 in d:
...     del d[1]
...
>>> d
{'uno': 1, 'dos': 2, 'tres': 3}
```

Comparar si dos diccionarios son iguales

En Python se puede utilizar el operador de igualdad `==` para comparar si dos diccionarios son iguales. Dos diccionarios son iguales si contienen el mismo conjunto de pares `clave: valor`, independientemente del orden que tengan.

Otro tipo de comparaciones entre diccionarios no están permitidas. Si se intenta, el intérprete lanzará la excepción `TypeError`.

```
>>> d1 = {'uno': 1, 'dos': 2}
>>> d2 = {'dos': 2, 'uno': 1}
>>> d3 = {'uno': 1}
>>> print(d1 == d2)
True
>>> print(d1 == d3)
False
>>> print(d1 > d2)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: '>' not supported between
instances of 'dict' and 'dict'
```

Diccionarios anidados en Python

Un diccionario puede contener un valor de cualquier tipo, entre ellos, otro diccionario. Este hecho se conoce como diccionarios anidados.

Para acceder al valor de una de las claves de un diccionario interno, se usa el operador de indexación anidada `[clave1][clave2]`...

Veámoslo con un ejemplo:

```
>>> d = {'d1': {'k1': 1, 'k2': 2}, 'd2':
{'k1': 3, 'k4': 4}}
>>> d['d1']['k1']
1
>>> d['d2']['k1']
3
>>> d['d2']['k4']
4
>>> d['d3']['k4']
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 'd3'
```

Obtener una lista con las claves de un diccionario

En ocasiones, es necesario tener almacenado en una lista las claves de un diccionario. Para ello, simplemente, pasa el diccionario como argumento del constructor `list()`. Las claves en la lista están en el mismo orden en que aparecen en el diccionario.

```
>>> d = {'uno': 1, 'dos': 2, 'tres': 3}
>>> list(d)
['uno', 'dos', 'tres']
```

Objetos vista de un diccionario

La clase `dict` implementa tres métodos muy particulares, dado que devuelven un tipo de dato, iterable, conocido como *objetos vista*. Estos objetos ofrecen una vista de las claves y valores contenidos en el diccionario y si el diccionario se modifica, dichos objetos se actualizan al instante.

Los métodos son los siguientes (ya los vimos al utilizar un diccionario junto con el bucle `for`):

- `keys()`: Devuelve una vista de las claves del diccionario.
- `values()`: Devuelve una vista de los valores del diccionario.
- `items()`: Devuelve una vista de pares (clave, valor) del diccionario.

```
>>> d = {'uno': 1, 'dos': 2, 'tres': 3}
# d.keys() es diferente a list(d), aunque ambos
# contengan las claves del diccionario
# d.keys() es de tipo dict_keys y list(d) es de
# tipo list
>>> v = d.keys()
>>> type(v)
<class 'dict_keys'>
>>> v
dict_keys(['uno', 'dos', 'tres'])

>>> l = list(d)
>>> type(l)
<class 'list'>
>>> l
['uno', 'dos', 'tres']

>>> v = d.values()
>>> type(v)
<class 'dict_values'>
>>> v
dict_values([1, 2, 3])

>>> v = d.items()
>>> type(v)
<class 'dict_items'>
>>> v
dict_items([('uno', 1), ('dos', 2), ('tres',
3)])
```