Colecciones básicas

Python trae incluidos de serie varios tipos de datos que representan colecciones de otros objetos. Una colección es un objeto que representa un grupo de objetos. También se les llama contenedores porque contienen otros objetos. Existen cuatro tipos básicos de colecciones:

- Listas - Colección de objetos que tienen un orden, está indexada, admite elementos duplicados y se puede modificar.
- Tuplas - Colección de objetos que tienen un orden, está indexada, admite elementos duplicados y no se puede modificar.
- Sets - Colección de objetos sin orden y sin índices. No permite elementos duplicados y se puede modificar.
- Dictionary - Colección sin orden que está indexada por claves que referencian a valores. No admite claves duplicadas pero si valores. Se pueden modificar.

Listas

Una lista es una colección ordenada de elementos del mismo tipo.

Crear una lista

En Python una lista se define entre corchetes (`[]`) separando los elementos por comas (`,`).

Como las listas suelen contener más de un elemento es una buena idea darlas un nombre en plural.

```
1  frutas = ['platano', 'manzana', 'naranja', 'limon']
2  print(type(frutas))
3  print(frutas)
```

> Resultado:

```
<class 'list'>
['platano', 'manzana', 'naranja', 'limon']
```

Acceder a un elemento

Dado que las listas están ordenadas podemos acceder a cada uno de sus elementos mediante su índice. Este índice será 0 para el primer elemento e irá incrementandose progresivamente.

Podemos utilizar un elemento de una lista de la misma forma que utilizamos una variable.

Para acceder a un elemento escribiremos el nombre de la lista y, a continuación, el índice del elemento al que deseamos acceder entre corchetes.

```
frutas = ['platano', 'manzana', 'naranja', 'limon']
print(frutas[0])
print(frutas[1])
print(type(frutas[1]))
```

> Resultado:

```
platano
manzana
<class 'str'>
```

Podemos acceder a los últimos elementos de una lista utilizando índices negativos de forma que el último elemento será el -1, el anteúltimo el -2 y así consecutivamente.

```
frutas = ['platano', 'manzana', 'naranja', 'limon']
print(frutas[-1])
print(frutas[-2])
```

> Resultado:

```
limon
naranja
```

Obtener el indice

Es posible obtener el índice de **la primera ocurrencia** de un elemento en la lista con el método `index`.

```
frutas = ['platano', 'coco', 'naranja', 'coco']
print(frutas.index('coco'))
```

> Resultado:

```
1
```

Obtener cuantos elementos

Es posible contar cuantas veces aparece un elemento en la lista con el método `count`.

Para obtener el número total de elementos en la lista utilizaremos la función `len`.

```
frutas = ['platano', 'coco', 'naranja', 'coco']
print("Ocurrencias coco:", frutas.count('coco'))
print("Total elementos:", len(frutas))
```

> Resultado:

```
Ocurrencias coco: 2
Total elementos: 4
```

> **Funciones vs Métodos**
>
> Cuando hablamos de un método de una colección, para invocarle utilizaremos el nombre de la variable de la colección, un punto, y el nombre del método seguido de los argumentos necesarios entre paréntesis (o ninguno si no es necesario). Un ejemplo es `count`.
>
> Cuando hablamos de una función, para invocarla, utilizaremos el nombre de la función y como argumento entre paréntesis irá el nombre de la variable de la colección. Un ejemplo es `len`.

Modificar un elemento

Para modificar un elemento utilizaremos el operador de asignación `=` de la misma forma que hacemos para modificar el valor de una variable.

```
frutas = ['platano', 'manzana', 'naranja', 'limon']
frutas[0] = 'coco'
print(frutas)
```

> Resultado:

```
['coco', 'manzana', 'naranja', 'limon']
```

Añadir un elemento

La forma más sencilla para añadir un elemento es con el método `append` que añade el nuevo elemento al final de la lista.

```
frutas = ['platano', 'manzana']
frutas.append('coco')
print(frutas)
```

> Resultado:

```
['platano', 'manzana', 'coco']
```

Crear una lista vacia

Podemos partir de una lista vacia y luego ir añadiendo elementos.

```
frutas = []
frutas.append('platano')
frutas.append('manzana')
frutas.append('coco')
print(frutas)
```

> Resultado:

```
['platano', 'manzana', 'coco']
```

Insertar un elemento

Es posible insertar un elemento en una determinada posición mediante el método `insert`. Pasaremos dos argumentos a este método: la posición que queremos para el nuevo elemento y cual es su valor.

```
frutas = ['platano', 'manzana']
frutas.insert(1, 'coco')
print(frutas)
```

> Resultado:

```
['platano', 'coco', 'manzana']
```

Concatenar dos listas

Con el operador `+` podemos concatenar dos listas.

```
f1 = ['platano', 'manzana']
f2 = ['coco', 'piña']
frutas = f1 + f2
print(frutas)
```

> Resultado:

```
['platano', 'manzana', 'coco', 'piña']
```

Extender una lista

Podemos añadir a una lista el contenido de otra lista mediante el método `extend` o el operador `+=`.

```
frutas = ['platano', 'manzana']
print(frutas)
frutas.extend(['coco', 'piña'])
print(frutas)
frutas += ['naranja', 'limon']
print(frutas)
```

> Resultado:

```
['platano', 'manzana']
['platano', 'manzana', 'coco', 'piña']
['platano', 'manzana', 'coco', 'piña', 'naranja', 'limon']
```

Eliminar elementos de una lista

Es posible eliminar un elemento por su posición o por su valor.

Por posicion

Para eliminar un elemento por su posición utilizaremos la función `del`. Cuando utilicemos `del` ya no será posible acceder al elemento borrado.

```
frutas = ['platano', 'manzana', 'naranja', 'limon']
del frutas[0]
print(frutas)
```

> Resultado:

```
['manzana', 'naranja', 'limon']
```

El método `pop` permite borrar el último elemento de la lista. Además, nos permite seguir trabajando con él una vez borrado.

```
frutas = ['platano', 'manzana', 'naranja', 'limon']
fruta_borrada = frutas.pop()
print(frutas)
print(fruta_borrada)
```

> Resultado:

```
['platano', 'manzana', 'naranja']
limon
```

Podemos incluir un argumento en el método `pop` para indicar el índice del elemento que deseamos borrar. También en este caso podemos seguir trabajando con el elemento.

```
frutas = ['platano', 'manzana', 'naranja', 'limon']
fruta_borrada = frutas.pop(1)
print(frutas)
print(fruta_borrada)
```

> Resultado:

```
['platano', 'naranja', 'limon']
manzana
```

Por valor

Si solo conocemos el valor del elemento que deseamos borrar pero no su posición podemos eliminarlo con el método `remove`.

```
frutas = ['platano', 'manzana', 'naranja', 'limon']
frutas.remove('naranja')
print(frutas)
```

> Resultado:

```
['platano', 'manzana', 'limon']
```

> **Remove elimina solo el primero**
>
> El método `remove` elimina solamente la primera ocurrencia del valor que especificamos. Si existe la posibilidad de que aparezca varias veces será necesario crear un bucle que se repita para borrar todos.

Eliminar todos los elementos

Con la función `clear` podemos eliminar todos los elementos de una lista.

```
frutas = ['platano', 'manzana', 'naranja', 'limon']
frutas.clear()
print(frutas)
print(len(frutas))
```

> Resultado:

```
[]
0
```

Ordenar una lista

Podemos ordenar los elementos de una lista con el método `sort`. Este método ordena la lista de forma permanente.

```
frutas = ['platano', 'manzana', 'naranja', 'limon']
frutas.sort()
print(frutas)
```

> Resultado:

```
['limon', 'manzana', 'naranja', 'platano']
```

Orden descendente

Podemos ordenar la lista en orden inverso pasando al método `sort` el argumento `reverse=True`.

```
frutas = ['platano', 'manzana', 'naranja', 'limon']
frutas.sort(reverse=True)
print(frutas)
```

> Resultado:

```
['platano', 'naranja', 'manzana', 'limon']
```

Orden temporal

Para ordenar una lista de forma solamente temporal podemos utilizar la función `sorted`. Esta función también admite como argumento `reverse=True`.

```
frutas = ['platano', 'manzana', 'naranja', 'limon']
print(frutas)
print(sorted(frutas))
print(frutas)
```

> Resultado:

```
['platano', 'manzana', 'naranja', 'limon']
['limon', 'manzana', 'naranja', 'platano']
['platano', 'manzana', 'naranja', 'limon']
```

Invertir el orden

Podemos invertir el orden de los elementos de una lista con el método `reverse`. Para volver a su estado original podemos invocar de nuevo el método `reverse`.

```
frutas = ['platano', 'manzana', 'naranja', 'limon']
frutas.reverse()
print(frutas)
```

> Resultado:

```
['limon', 'naranja', 'manzana', 'platano']
```

Comprobar la existencia

Podemos comprobar si existe un elemento en la lista con un `if`.

```
frutas = ['platano', 'manzana', 'naranja']
if 'manzana' in frutas:
    print('manzana está en la lista')
if 'coco' in frutas:
    print('coco esta en la lista')
```

> Resultado:

```
manzana está en la lista
```

Recorrer una lista

Es posible recorrer los elementos de una lista para realizar operaciones sobre cada uno de ellos. Para recorrer los elementos de una lista utilizaremos un bucle `for`.

```
frutas = ['platano', 'manzana', 'naranja']
for fruta in frutas:
    print(fruta)
```

> Resultado:

```
platano
manzana
naranja
```

Poniendo `for fruta in frutas:` logramos que las sentencias que vienen a continuación (están desplazadas cuatro espacios a la derecha) se repitan para cada uno de los elementos de la lista. En este caso solo hay una sentencia: `print(fruta)` que se repetirá para cada una de las frutas. Podiamos enunciar este código como: "Para cada fruta en la lista de frutas, imprime el nombre de la fruta".

`fruta` es el nombre de variable que utilizamos para poder acceder al elemento correspondiente en cada repetición del bucle. Podemos utilizar el nombre que consideremos

más conveniente.

Función "range"

La función `range` permite crear facilmente una serie de números.

```
for numero in range(1, 5):
    print(numero)
```

> Resultado:

```
1
2
3
4
```

La función `range` genera una serie de números comenzando con el primer número que pasamos como argumento entre paréntesis y finaliza cuando alcanza el segundo número (sin incluirle).

Si pasamos solamente un argumento, `range` lo considerará el valor de parada y comenzará la secuencia en 0, así `range(3)` genera la serie [0, 1, 2].

Si pasamos un tercer argumento podemos especificar un salto entre los elementos.

```
for numero in range(1, 10, 3):
    print(numero)
```

> Resultado:

```
1
4
7
```

Función "list"

Podemos transformar otros objetos en listas con la función `list`. Es posible obtener, de esta forma, una lista de números a partir de `range`.

```
rango = range(1, 5)
print(type(rango))
lista = list(rango)
print(type(lista))
print(lista)
```

> Resultado:

```
<class 'range'>
<class 'list'>
[1, 2, 3, 4]
```

Estadísticos básicos

Las funciones `min`, `max` y `sum` nos ofrecen unos estadisticos básicos de una lista numérica.

```
numeros = list(range(0, 10))
print(min(numeros))
print(max(numeros))
print(sum(numeros))
```

> Resultado:

```
0
9
45
```

Comprensión de listas

Podemos generar una lista a partir de un rango de forma manual partiendo de una lista vacia.

```
cubos = []
for numero in range(1, 5):
    cubo = numero ** 3
    cubos.append(cubo)
print(cubos)
```

> Resultado:

```
[1, 8, 27, 64]
```

Una comprensión de lista o "list comprehension" en inglés nos permite generar una lista de forma compacta y simple. Veamos el equivalente al ejemplo anterior utilizando compresión de listas para crear nuestra lista.

```
cubos = [numero**3 for numero in range(1, 5)]
print(cubos)
```

> Resultado:

```
[1, 8, 27, 64]
```

Acceder a parte de una lista

A un subconjunto de los elementos de una lista se le denomina "slice" en inglés que podemos traducir por rebanada, trozo o corte. A la operación de obtener estas rebanadas se la denomina "slicing" que denominaremos cortar una lista. El corte de una lista es, a su vez, otra lista.

Para obtener un corte de una lista debemos especificar el índice del primer y el último elemento. Python no incluirá el último elemento especificado en el corte.

```
frutas = ['platano', 'manzana', 'naranja', 'limon', 'coco']
corte = frutas[1:3]
print(type(corte))
print(corte)
```

> Resultado:

```
<class 'list'>
['manzana', 'naranja']
```

Si omitimos el primer índice, Python comienza por el primer elemento de la lista. Si omitimos el segundo índice, Python cortará hasta el final de la lista.

```
frutas = ['platano', 'manzana', 'naranja', 'limon', 'coco']
print(frutas[:3])
print(frutas[1:])
```

> Resultado:

```
['platano', 'manzana', 'naranja']
['manzana', 'naranja', 'limon', 'coco']
```

Podemos utilizar valores negativos de los índices para acceder a posiciones desde el final de la lista. -1 sería el último elemento, -2 sería el anteúltimo elemento y así progresivamente. De esta forma podemos acceder, por ejemplo, a los tres últimos elementos de una lista sin conocer su tamaño:

```
frutas = ['platano', 'manzana', 'naranja', 'limon', 'coco']
print(frutas[-3:])
```

> Resultado:

```
['naranja', 'limon', 'coco']
```

En el corte podemos especificar un tercer valor que indique el salto entre elementos.

```

```

```
2
   frutas = ['platano', 'manzana', 'naranja', 'limon', 'coco']
   print(frutas[1:4:2])
```

> Resultado:

```
['manzana', 'limon']
```

Copiar una lista

Podemos realizar una copia de una lista (crear una lista nueva con los mismos elementos que la original) realizando un corte que incluya a todos los elementos, es decir, que no especifique ninguno de los índices.

```
1  frutas = ['platano', 'manzana', 'naranja']
2  copia_frutas = frutas[:]
3  copia_frutas.append('coco')
4  print("Original:", frutas)
5  print("Copia:", copia_frutas)
```

> Resultado:

```
Original: ['platano', 'manzana', 'naranja']
Copia: ['platano', 'manzana', 'naranja', 'coco']
```

Otra opción para copiar una lista es utilizar el método `copy`.

```
1  frutas = ['platano', 'manzana', 'naranja']
2  copia_frutas = frutas.copy()
3  copia_frutas.append('coco')
4  print("Original:", frutas)
5  print("Copia:", copia_frutas)
```

> Resultado:

```
Original: ['platano', 'manzana', 'naranja']
Copia: ['platano', 'manzana', 'naranja', 'coco']
```

Métodos de listas

Esta es la relación por orden alfabético de algunos de los métodos disponibles en un lista.

Método	Descripción
append	Añade un elemento al final de la lista
clear	Elimina todos los elementos de la lista
copy	Devuelve una copia de la lista
count	Devuelve el número de elementos que contienen un determinado valor
extend	Añade los elementos de una lista (o de cualquier iterable) al final de la lista
index	Devuelve el índice del primer elemento con un determinado valor
insert	Añade un elemento en la posición especificada
pop	Elimina un elemento en la posición especificada
remove	Elimina un elemento con un determinado valor
reverse	Invierte el orden de los elementos de una lista
sort	Ordena los elementos de la lista

Tuplas

Una tupla `Tuple` es una lista inmutable, es decir, una lista que no puede cambiar a lo largo de la ejecución de un programa. Por ello no podemos añadir ni eliminar elementos de una tupla una vez creada.

Una tupla puede contener una mezcla de diferente tipos de datos.

Una tupla se crea igual que una lista, pero en lugar de corchetes utilizaremos paréntesis.

```
1  posicion_tesoro = (37.0083, -3.6)
2  print("Latitud:", posicion_tesoro[0])
3  print("Longitud:", posicion_tesoro[1])
```

> Resultado:

```
Latitud: 37.0083
Longitud: -3.6
```

Si bien no es posible cambiar uno de los valores de una tupla, si es posible asignar una nueva tupla a la variable que la contiene, en el ejemplo anterior sería posible poner `posicion_tesoro`

`= (27.745003, -82.759659)`.

Técnicamente una tupla se define por la presencia de las comas. Los paréntesis solo las hacen más legibles. Así, si quisieramos definir una tupla de un solo elemento pondríamos: `t = (14,)`.

Función "tuple"

Con la función `tuple` podemos crear tuplas a partir de otras colecciones (objetos que implementen el protocolo `Iterable`).

```
frutas = ['platano', 'manzana', 'naranja']
print(type(frutas))
t = tuple(frutas)
print(t)
print(type(t))
```

> Resultado:

```
<class 'list'>
('platano', 'manzana', 'naranja')
<class 'tuple'>
```

Conjuntos

Un conjunto o `Set` es una colección sin orden de objetos inmutables que no admite duplicados.

Un `Set` se define entre `{}`.

```
frutas = {'coco', 'platano', 'manzana', 'coco', 'naranja', 'coco'}
print(type(frutas))
print(frutas)
```

> Resultado:

```
<class 'set'>
{'manzana', 'coco', 'naranja', 'platano'}
```

El 'coco' solo se ha añadido una vez al `Set`. Dado que los elementos no tienen orden, no podemos utilizar un índice entre corchetes para acceder a ellos.

Función "set"

Con la función `set` podemos crear conjuntos a partir de otras colecciones (objetos que implementen el protocolo `Iterable`).

```
frutas = ('platano', 'manzana', 'naranja')
print(type(frutas))
frutas_set = set(frutas)
print(type(frutas_set))
print(frutas_set)
```

> Resultado:

```
<class 'tuple'>
<class 'set'>
{'platano', 'manzana', 'naranja'}
```

Acceder a los elementos

Es posible iterar sobre los elementos de un `Set` mediante un bucle `for`.

```
frutas = {'platano', 'manzana', 'naranja'}
for fruta in frutas:
    print(fruta)
```

> Resultado:

```
platano
naranja
manzana
```

Comprobar la existencia

Podemos comprobar si un elemento existe en la colección con la palabra clave `in`. Devolverá `True` o `False` dependiendo de si existe o no.

```
frutas = {'platano', 'manzana', 'naranja'}
print('platano' in frutas)
print('coco' in frutas)
print('coco' not in frutas)
```

> Resultado:

```
True
False
True
```

Añadir elementos

Es posible añadir un nuevo elemento al `Set` con el método `add`. Con el método `update` podemos añadir varios elementos al `Set` pasandole como argumento una colección.

```
frutas = {'platano', 'manzana', 'naranja'}
frutas.add('pera')
frutas.update(['coco', 'piña', 'platano'])
print(frutas)
```

> Resultado:

```
{'pera', 'coco', 'manzana', 'piña', 'naranja', 'platano'}
```

Eliminar elementos

Para eliminar elementos utilizaremos los métodos `remove` y `discard`. Ambos eliminan el elemento que les pasamos como argumento. La diferencia entre ambos métodos es que `remove` lanza un error si el elemento no existe y `discard` no.

```
frutas = {'platano', 'manzana', 'naranja'}
frutas.remove('platano')
#frutas.remove('coco')   # Produciría un error
frutas.discard('coco')   # No produce error
print(frutas)
```

> Resultado:

```
{'manzana', 'naranja'}
```

Para borrar todos los elementos de la colección utilizaremos el método `clear`.

```
frutas = {'platano', 'manzana', 'naranja'}
frutas.clear()
print(frutas)
```

> Resultado:

```
set()
```

Estadísticos básicos

La función `len` nos devuelve el tamaño de un `Set`. Las funciones `min`, `max` y `sum` nos ofrecen unos estadisticos básicos de una colección de números.

Si los valores son strings las funciones `len`, `min` y `max` nos devolverán el tamaño de la colección y los string menor y mayor según el orden alfabetico.

```
numeros = {12, 8, 20, 20}
print(len(numeros))
print(min(numeros))
print(max(numeros))
print(sum(numeros))
```

> Resultado:

```
3
8
20
40
```

Operaciones de conjuntos

Un conjunto `Set` admite las operaciones clásicas de teoría de conjuntos (unión, intersección, diferencia y diferencia simétrica). Veamos un ejemplo:

```
s1 = {'platano', 'manzana', 'naranja'}
s2 = {'coco', 'piña', 'platano'}
print('s1:', s1)
print('s2:', s2)
print('Unión:', s1 | s2)
print('Intersección:', s1 & s2)
print('Diferencia:', s1 - s2)
print('Diferencia simétrica:', s1 ^ s2)
```

> Resultado:

```
s1: {'platano', 'manzana', 'naranja'}
s2: {'coco', 'platano', 'piña'}
Unión: {'platano', 'piña', 'manzana', 'naranja', 'coco'}
Intersección: {'platano'}
Diferencia: {'manzana', 'naranja'}
Diferencia simétrica: {'manzana', 'coco', 'piña', 'naranja'}
```

Es posible utilizar la versión en método de los operadores. El código anterior sería equivalente a escribir:

```
print('Unión:', s1.union(s2))
print('Intersección:', s1.intersection(s2))
print('Diferencia:', s1.difference(s2))
print('Diferencia simétrica:', s1.symmetric_difference(s2))
```

Métodos de conjuntos

Esta es la relación por orden alfabético de algunos de los métodos disponibles en un conjunto.

Método	Descripción
add	Añade un elemento al conjunto
clear	Elimina todos los elementos del conjunto
copy	Devuelve una copia del conjunto
difference	Devuelve un conjunto que contiene la diferencia entre dos o más conjuntos
difference_update	Elimina los elementos de un conjunto que están incluidos en otro conjunto
discard	Elimina el elemento especificado
intersection	Devuelve un conjunto que es la intersección de otros dos conjuntos
intersection_update	Elimina los elementos de un conjunto que no están incluidos en otros conjuntos
isdisjoint	Devuelve cuando dos conjuntos tienen intersección o no
issubset	Devuelve cuando otro conjunto contiene el conjunto o no
issuperset	Devuelve cuando el conjunto contiene a otro conjunto o no
pop	Elimina un elemento del conjunto
remove	Elimina el elemento especificado
symmetric_difference	Devuelve un conjunto con la diferencia simétrica de dos conjuntos
symmetric_difference_update	Inserta la diferencia simétrica de un conjunto en otro
union	Devuelve un conjunto con la unión de conjuntos
update	Actualiza un conjunto con la unión del conjunto con otros

Diccionarios

Un diccionario (`Dictionary`) es un conjunto de asociaciones entre una clave y un valor.

Un diccionario se crea usando llaves `{ }` y dentro de estas, separadas por comas, parejas `clave:valor`.

```
ciudades = {'España': 'Madrid',
            'Mexico': 'Guadalajara',
            'Paraguay': 'Asuncion',
            'Costa Rica': 'Cartago',
            'Argentina': 'Mendoza'}
print(ciudades)
print(len(ciudades))
```

> Resultado:

```
{'España': 'Madrid', 'Mexico': 'Guadalajara', 'Paraguay': 'Asuncion',
'Costa Rica': 'Cartago', 'Argentina': 'Mendoza'}
5
```

Con la función `len` podemos consultar cuantas entradas existen en nuestro diccionario.

Acceder a un elemento

Es posible acceder a los elementos escribiendo su clave entre corchetes. También se puede utilizar el método get.

```
ciudades = {'España': 'Madrid',
            'Mexico': 'Guadalajara',
            'Costa Rica': 'Cartago'}
print(ciudades['España'])
print(ciudades.get('Mexico'))
```

> Resultado:

```
Madrid
Guadalajara
```

El método `get` admite un segundo argumento opcional que será devuelto si la clave buscada no existe. Si existen posibilidades de que la clave buscada no exista es recomendable utilizar `get`. Si no especificamos el segundo argumento en el `get` si la clave no existe devuelve el valor `None`.

```
ciudades = {'España': 'Madrid',
            'Mexico': 'Guadalajara',
            'Costa Rica': 'Cartago'}
print(ciudades.get('Italia', 'El pais no está registrado'))
print(ciudades.get('Italia'))
```

> Resultado:

```
El pais no está registrado
None
```

Añadir un nuevo elemento

Para añadir un nuevo elemento utilizaremos la clave entre corchetes y especificaremos cual es su valor.

```
ciudades = {'España': 'Madrid',
            'Costa Rica': 'Cartago'}
ciudades['Argentina'] = 'Mendoza'
print(ciudades)
```

> Resultado:

```
{'España': 'Madrid', 'Costa Rica': 'Cartago', 'Argentina': 'Mendoza'}
```

Cambiar un valor

El valor asociado con cada clave se puede cambiar.

```
ciudades = {'España': 'Madrid',
            'Costa Rica': 'Cartago'}
ciudades['España'] = 'Santander'
print(ciudades)
```

> Resultado:

```
{'España': 'Santander', 'Costa Rica': 'Cartago'}
```

Eliminar elementos

Existen tres formas de eliminar elementos de un diccionario:

- Utilizando el método `pop(<key>)` eliminamos la entrada con la clave especificada en `key`. Este método devuelve el valor de la clave que se desea eliminar. Si la clave no existe un valor por defecto (si se ha asignado usando `setdefault`) es devuelto. Si no se ha asignado, se generará un error.
- El método `popitem()` elimina el último item insertado en el diccionario. El método devuelve la pareja `key:value` eliminada.
- La palabra clave `del` elimina la entrada con la clave especificada sin devolver nada. Puede ser más eficiente que `pop(<key>)`.

Veamos un ejemplo de cada uno de ellos:

```
ciudades = {'España': 'Madrid',
            'Mexico': 'Guadalajara',
            'Costa Rica': 'Cartago'}
print(ciudades)
ciudades.popitem() # Borra 'Costa Rica'
print(ciudades)
ciudades.pop('España')
print(ciudades)
del ciudades['Mexico']
print(ciudades)
```

> Resultado:

```
{'España': 'Madrid', 'Mexico': 'Guadalajara', 'Costa Rica': 'Cartago'}
{'España': 'Madrid', 'Mexico': 'Guadalajara'}
{'Mexico': 'Guadalajara'}
{}
```

Con el método `clear` podemos eliminar todas las entradas del diccionario.

```
ciudades = {'España': 'Madrid',
            'Mexico': 'Guadalajara',
            'Costa Rica': 'Cartago'}
print(ciudades)
ciudades.clear()
print(ciudades)
```

> Resultado:

```
{'España': 'Madrid', 'Mexico': 'Guadalajara', 'Costa Rica': 'Cartago'}
{}
```

Recorrer un diccionario

Podemos recorrer un diccionario con un ciclo `for`. El ciclo procesa cada una de las claves del diccionario con las cuales podemos acceder a los valores. Veamos un ejemplo:

```
ciudades = {'España': 'Madrid',
            'Mexico': 'Guadalajara',
            'Costa Rica': 'Cartago'}
for pais in ciudades:
    print(pais, end=', ')
    print(ciudades[pais])
```

> Resultado:

```
España, Madrid
Mexico, Guadalajara
Costa Rica, Cartago
```

El ejemplo anterior es equivalente a haber escrito `for pais in ciudades.keys():`. Si solo especificamos el nombre del diccionario se itera por las claves, pero podemos iterar también por los valores y por los pares (clave-valor).

Para poder iterar sobre los valores directamente utilizaremos el método `values` del diccionario.

```
ciudades = {'España': 'Madrid',
            'Mexico': 'Guadalajara',
            'Costa Rica': 'Cartago'}
for ciudad in ciudades.values():
    print(ciudad)
```

> Resultado:

```
Madrid
Guadalajara
Cartago
```

Es posible iterar por las entradas (clave-valor) simultaneamente con el método `items`.

```
ciudades = {'España': 'Madrid',
            'Mexico': 'Guadalajara',
            'Costa Rica': 'Cartago'}
for pais, ciudad in ciudades.items():
    print(f"{pais}: {ciudad}")
```

> Resultado:

```
España: Madrid
Mexico: Guadalajara
Costa Rica: Cartago
```

Podemos iterar por las claves o por los valores en orden utilizando una expresión como `for k in sorted(ciudades.keys()):`.

También podemos hacerlo evitando repeticiones si lo convertimos en un conjunto (`Set`) previamente así `for v in set(ciudades.values()):`.

Valores, claves y elementos

En un diccionario hay tres métodos que obtienen vistas diferentes de los datos que contiene:

- El método `values` devuelve un vista de los valores.
- El método `keys` devuelve una vista de las claves.
- El método `items` devuelve una vista de los elementos del diccionario (pares clave-valor).

```
ciudades = {'España': 'Madrid',
            'Mexico': 'Guadalajara'}
print(ciudades.values())
print(ciudades.keys())
print(ciudades.items())
```

> Resultado:

```
dict_values(['Madrid', 'Guadalajara'])
dict_keys(['España', 'Mexico'])
dict_items([('España', 'Madrid'), ('Mexico', 'Guadalajara')])
```

Comprobar la existencia

Podemos comprobar si una clave pertenece al diccionario con la palabra clave `in` o `not in`.

```
ciudades = {'España': 'Madrid',
            'Mexico': 'Guadalajara'}
print('Mexico' in ciudades)
print('Argentina' in ciudades)
print('Argentina' not in ciudades)
```

> Resultado:

```
True
False
True
```

Anidar diccionarios

Las claves y los valores de un diccionario pueden ser cualquier objeto. Esto nos permite incluir en los valores de un diccionario listas, tuplas, conjuntos o incluso otros diccionarios. Veamos un ejemplo:

```
estaciones = {'Primavera': ('Marzo', 'Abril', 'Mayo'),
              'Verano': ('Junio', 'Julio', 'Agosto'),
              'Otoño': ('Septiembre', 'Octubre', 'Noviembre'),
              'Invierno': ('Diciembre', 'Enero', 'Febrero')}
print(estaciones['Primavera'])
print(estaciones['Primavera'][1])
```

> Resultado:

```
('Marzo', 'Abril', 'Mayo')
Abril
```

Métodos de diccionarios

Esta es la relación por orden alfabético de algunos de los métodos disponibles en un diccionario.

Método	Descripción
clear	Elimina todos los elementos del diccionario.
copy	Devuelve una copia del diccionario.
fromkeys	Devuelve un diccionario con las claves y valores especificados.
get	Devuelve el valor para la clave especificada.
items	Devuelve una lista que contiene una tupla para cada pareja clave-valor.
keys	Devuelve una lista con las claves del diccionario.
pop	Elimina el elemento con la clave especificada.
popitem	Elimina la última entrada insertada.
setdefault	Devuelve el valor de la clave especificada. Si la clave no existe la crea.
update	Actualiza el diccionario con los pares clave-valor especificados.
values	Devuelve una lista con todos los valores del diccionario.