## Tuplas en Python

La instrucción tuple en Python es un tipo contenedor, compuesto, que en un principio se pensó para almacenar grupos de elementos heterogéneos, aunque también puede contener elementos homogéneos.

Junto a las clases list y range, es uno de los tipos de secuencia en Python, con la particularidad de que son inmutables. Esto último quiere decir que su contenido NO se puede modificar después de haber sido creada.

En general, para crear una tupla en Python simplemente hay que definir una secuencia de elementos separados por comas.

Por ejemplo, para crear una tupla con los números del 1 al 5 se haría del siguiente modo:

**numeros = 1, 2, 3, 4, 5**

tuple también puede almacenar elementos de distinto tipo:

**elementos = 3, 'a', 8, 7.2, 'hola'**

Incluso pueden contener otros elementos compuestos y objetos, como listas, otras tuplas, etc.:

**tup = 1, ['a', 'e', 'i', 'o', 'u'], 8.9, 'hola'**

Diferentes formas que existen de crear una tupla en Python:

Para crear una tupla vacía, usa paréntesis () o el constructor de la clase tuple() sin parámetros.

Para crear una tupla con un único elemento: elem, o (elem, ). Observe que siempre se añade una coma.

Para crear una tupla de varios elementos, sepáralos con comas: a, b, c o (a, b, c).

Las tuplas también se pueden crear usando el constructor tuple(iterable).

En este caso, el constructor crea una tupla cuyos elementos son los mismos y están en el mismo orden que los ítems del iterable.

Lo que determina que una secuencia de elementos sea una tupla es la coma **,** no los paréntesis. Los paréntesis son opcionales y solo se necesitan para crear una tupla vacía o para evitar ambigüedades.

Para acceder a un elemento de una tupla se utilizan los índices. **Un índice es un número entero que indica la posición de un elemento en una tupla**. El primer elemento de una tupla siempre comienza enel índice 0.

Por ejemplo, en una tupla con 3 elementos, los índices de cada uno de los ítems serían 0, 1 y 2.

```
tupla = ('a','b', 'd')
tupla[0] # Primer elemento de la tupla. Índice 0
'a'
tupla[1] # Segundo elemento de la tupla. Índice 1'b'
```

Si se intenta acceder a un índice que está fuera del rango de la tupla, el intérprete lanzará la excepción `IndexError`. De igual modo, si se utiliza un índice que no es un número entero, se lanzará la excepción `TypeError`:

```
tupla = 1, 2, 3 # Los índices válidos son 0, 1 y 2
tupla[8]
Traceback (most recent call last):
File "<input>", line 1, in <module>
IndexError: tuple index out of range
tupla[1.0]
Traceback (most recent call last):
File "<input>", line 1, in <module>
TypeError: tuple indices must be integers or slices, not float
```

Al igual que ocurre con las listas (y todos los tipos secuenciales), está permitido usar índices negativos para acceder a los elementos de una tupla. En este caso, el índice -1 hace referencia al último elemento de la secuencia, el -2 al penúltimo y así, sucesivamente:

```
bebidas = ('agua', 'café', 'batido', 'sorbete')
bebidas[-1]
'sorbete'
bebidas[-3]
'café'
```

También es posible acceder a un subconjunto de elementos de una tupla utilizando el operador `[:]`:

```
vocales = 'a', 'e', 'i', 'o', 'u'
vocales[2:3] # Elementos desde el índice 2 hasta el índice 3-1('i',)
vocales[2:4] # Elementos desde el 2 hasta el índice 4-1
('i', 'o')
vocales[:] # Todos los elementos
('a', 'e', 'i', 'o', 'u')
vocales[1:] # Elementos desde el índice 1
('e', 'i', 'o', 'u')
vocales[:3] # Elementos hasta el índice 3-1
('a', 'e', 'i')
```

O indicando un salto entre los elementos con el operador `[::]`:

```
pares = 2, 4, 6, 8, 10, 12, 14
pares[::2] # Acceso a los elementos de 2 en 2
(2, 6, 10, 14)
pares[1:5:2] # Elementos del índice 1 al 4 de 2 en 2
(4, 8)
pares[1:6:3] # Elementos del índice 1 al 5 de 3 en 3
(4, 10)
```

El bucle for en Python es una de las estructuras ideales para iterar sobre los elementos de una secuencia. Para recorrer una tupla en Python utiliza la siguiente estructura:

```
colores = 'azul', 'blanco', 'negro'
for color in colores:
    print(color)
azul
blanco
negro
```

Como cualquier tipo secuencia, para conocer la longitud de una tupla en Python se hace uso de la función `len()`. Esta función devuelve el número de elementos de una tupla:

```
vocales = ('a', 'e', 'i', 'o', 'u')
len(vocales)
5
```

| Funciones | Descripción |
|---|---|
| `enumerate()` | devuelve el índice y el valor de todos los elementos de la tupla como una tupla. Devuelve un objeto `enumerate` |
| `len()` | devuelva el número de artículos de una tupla o la longitud de la tupla. |
| `tuple()` | convertir una secuencia (tupla, conjunto, cadena, diccionario) en tupla. |
| `max()` | devuelve el número máximo de la tupla. |
| `min()` | devuelva el número mínimo de la tupla. |
| `sorted()` | devolver una tupla ordenada. |
| `sum()` | devuelve la suma de todos los elementos de la tupla. |

# Listas en Python

Las listas en Python son un tipo **contenedor**, compuesto, que **se usan para almacenar conjuntos de elementos relacionados** del mismo tipo o de tipos distintos.

Junto a las intrucciones *tuple*, *range* y *str*, **son uno de los tipos de secuencia en Python**, con la particularidad de que son *mutables*. Esto último quiere decir que su contenido se puede modificar después de haber sido creada.

Para crear una lista en Python, simplemente hay que encerrar una secuencia de elementos separados por comas entre paréntesis cuadrados `[]`.

Por ejemplo, para crear una lista con los números del 1 al 10 se haría del siguiente modo:

```
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Las listas pueden almacenar elementos de distinto tipo. La siguiente lista también es válida:

```
elementos = [3, 'a', 8, 7.2, 'hola']
```

Incluso pueden contener otros elementos compuestos, como objetos u otras listas:

```
lista = [1, ['a', 'e', 'i', 'o', 'u'], 8.9, 'hola']
```

Las listas también se pueden crear usando el constructor de la clase, `list(iterable)`.

En este caso, el constructor crea una lista cuyos elementos son los mismos y están en el mismo orden que los ítems del iterable.

Por ejemplo, el tipo *str* también es un tipo secuencia. Si pasamos un string al constructor `list()` creará una lista cuyos elementos son cada uno de los caracteres de la cadena:

```
vocales = list('aeiou') vocales
['a', 'e', 'i', 'o', 'u']
```

```
Dos alternativas de crear una lista vacía: lista_1
= [] # Opción 1
lista_2 = list() # Opción 2
```

Para acceder a un elemento de una lista se utilizan los índices. **Un índice es un número entero que indica la posición de un elemento en una lista**. El primer elemento de una lista siempre comienza en el índice0.

```
Por ejemplo, en una lista con 4 elementos, los índices de cada uno de los ítems serían 0, 1, 2 y 3.
lista = ['a', 'b', 'd', 'i', 'j']
lista[0] # Primer elemento de la lista. Índice 0
'a'
lista[3] # Cuarto elemento de la lista. Índice 3'i'
```

Si se intenta acceder a un índice que está fuera del rango de la lista, el intérprete lanzará la excepción `IndexError`. De igual modo, si se utiliza un índice que no es un número entero, se lanzará la excepción `TypeError`:

```
lista = [1, 2, 3] # Los índices válidos son 0, 1 y 2
lista[8]
Traceback (most recent call last):
File "<input>", line 1, in <module>
IndexError: list index out of range
lista[1.0]
Traceback (most recent call last):
File "<input>", line 1, in <module>
TypeError: list indices must be integers or slices, not float
```

Las listas pueden contener otros elementos de tipo secuencia de forma anidada. Por ejemplo, una lista que uno de sus ítems es otra lista. Del mismo modo, se puede acceder a los elementos de estos tipos usando índices compuestos o anidados:

```
lista = ['a', ['d', 'b'], 'z']
lista[1][1] # lista[1] hace referencia a la lista anidada
'b'
```

En Python está permitido usar índices negativos para acceder a los elementos de una secuencia. En este caso, el índice -1 hace referencia al último elemento de la secuencia, el -2 al penúltimo y así, sucesivamente:

```
vocales = ['a', 'e', 'i', 'o', 'u']
vocales[-1]
'u'
vocales[-4]
'e'
```

También es posible acceder a un subconjunto de elementos de una lista utilizando rangos en los índices. Esto es usando el operador `[:]`:

```
vocales = ['a', 'e', 'i', 'o', 'u']
vocales[2:3] # Elementos desde el índice 2 hasta el índice 3-1 ['i']
vocales[2:4] # Elementos desde el 2 hasta el índice 4-1
['i', 'o']
vocales[:] # Todos los elementos
['a', 'e', 'i', 'o', 'u']
vocales[1:] # Elementos desde el índice 1
['e', 'i', 'o', 'u']
vocales[:3] # Elementos hasta el índice 3-1
['a', 'e', 'i']
```

También es posible acceder a los elementos de una lista indicando un paso con el operador `[::]`:

```
letras = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k']
letras[::2] # Acceso a los elementos de 2 en 2['a',
'c', 'e', 'g', 'i', 'k']
letras[1:5:2] # Elementos del índice 1 al 4 de 2 en 2
['b', 'd']
letras[1:6:3] # Elementos del índice 1 al 5 de 3 en 3
```

```
['b', 'e']
```

Ya hemos visto que se puede usar el bucle for en Python para recorrer los elementos de una secuencia. En nuestro caso, para recorrer una lista en Python utilizaríamos la siguiente estructura:

```
colores = ['azul', 'blanco', 'negro']
for color in colores:
    print(color)
azul
blanco
negro
```

Las listas son secuencias mutables, es decir, sus elementos pueden ser modificados (se pueden añadir nuevos ítems, actualizar o eliminar).

Para añadir un nuevo elemento a una lista se utiliza el método `append()` y para añadir varios elementos, el método `extend()`:

```
vocales = ['a']
vocales.append('e') # Añade un elemento
vocales
['a', 'e']
vocales.extend(['i', 'o', 'u']) # Añade un grupo de elementos
vocales
['a', 'e', 'i', 'o', 'u']
```

También es posible utilizar el operador de concatenación `+` para unir dos listas en una sola. El resultado es una nueva lista con los elementos de ambas:

```
lista_1 = [1, 2, 3]
lista_2 = [4, 5, 6]
nueva_lista = lista_1 + lista_2
nueva_lista
[1, 2, 3, 4, 5, 6]
```

Por otro lado, el operador `*` repite el contenido de una lista `n` veces:

```
numeros = [1, 2, 3]
numeros *= 3
numeros
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Y para terminar esta sección, indicarte que también es posible añadir un elemento en una posición concreta de una lista con el método `insert(índice, elemento)`. Los elementos cuyo índice sea mayor a `índice` se desplazan una posición a la derecha:

```
vocales = ['a', 'e', 'u']
vocales.insert(2, 'i')
vocales
['a', 'e', 'i', 'u']
```

Es posible modificar un elemento de una lista en Python con el operador de asignación `=`. Para ello, lo único que necesitas conocer es el índice del elemento que quieres modificar o el rango de índices: `vocales = ['o', 'o', 'o', 'o', 'u']`

```
# Actualiza el elemento del índice 0
```

```
vocales[0] = 'a'
vocales
['a', 'o', 'o', 'o', 'u']
# Actualiza los elementos entre las posiciones 1 y 2vocales[1:3]
= ['e', 'i']
vocales
['a', 'e', 'i', 'o', 'u']
```

En Python se puede eliminar un elemento de una lista de varias formas.
Con la sentencia `del` se puede eliminar un elemento a partir de su índice:

```
# Elimina el elemento del índice 1
vocales = ['a', 'e', 'i', 'o', 'u']
del vocales[1]
vocales
['a', 'i', 'o', 'u']
# Elimina los elementos con índices 2 y 3
vocales = ['a', 'e', 'i', 'o', 'u']
del vocales[2:4]
vocales
['a', 'e', 'u']
# Elimina todos los elementos
del vocales[:]
vocales
[]
```

Además de la sentencia `del`, podemos usar los métodos `remove()` y `pop([i])`. `remove()` elimina la primera ocurrencia que se encuentre del elemento en una lista. Por su parte, `pop([i])` obtiene el elemento cuyo índice sea igual a `i` y lo elimina de la lista. Si no se especifica ningún índice, recupera y elimina el último elemento.

```
letras = ['a', 'b', 'k', 'a', 'v']
# Elimina la primera ocurrencia del carácter a
letras.remove('a')
letras
['b', 'k', 'a', 'v']
# Obtiene y elimina el último elemento
letras.pop()
'v'
letras
['b', 'k', 'a']
```

Finalmente, es posible eliminar todos los elementos de una lista a través del método `clear()`: `letras = ['a', 'b', 'c']`

```
letras.clear()
letras
[]
```

El código anterior sería equivalente a `del letras[:]`.

Como cualquier tipo secuencia, para conocer la longitud de una lista en Python se hace uso de la función `len()`. Esta función devuelve el número de elementos de una lista:

```
vocales = ['a', 'e', 'i', 'o', 'u']
```

```
len(vocales)
5
```

| Método | Descripción |
|---|---|
| `append()` | Añade un nuevo elemento al final de la lista. |
| `extend()` | Añade un grupo de elementos (iterables) al final de la lista. |
| `insert(indice, elemento)` | Inserta un elemento en una posición concreta de la lista. |
| `remove(elemento)` | Elimina la primera ocurrencia del elemento en la lista. |
| `pop([i])` | Obtiene y elimina el elemento de la lista en la posición i. Si no se especifica, obtiene y elimina el último elemento. |
| `clear()` | Borra todos los elementos de la lista. |
| `index(elemento)` | Obtiene el índice de la primera ocurrencia del elemento en la lista. Si el elemento no se encuentra, se lanza la excepción `ValueError`. |
| `count(elemento)` | Devuelve el número de ocurrencias del elemento en la lista. |

| Método | Descripción |
| --- | --- |
| `sort()` | Ordena los elementos de la lista utilizando el operador `<`. |
| `reverse()` | Obtiene los elementos de la lista en orden inverso. |
| `copy()` | Devuelve una copia poco profunda de la lista. |

## Conjuntos (set) en Python

El tipo set en Python es utilizado para trabajar con conjuntos de elementos. La principal característica de este tipo de datos es que es una colección cuyos elementos no guardan ningún orden y que además son únicos.

Estas características hacen que los principales usos de esta clase sean conocer si un elemento pertenece o no a una colección y eliminar duplicados de un tipo secuencial (*list*, *tuple* o *str*).

Además, esta clase también implementa las típicas operaciones matemáticas sobre conjuntos: *unión*, *intersección*, *diferencia*, ⋯

Para crear un conjunto, basta con encerrar una serie de elementos entre llaves {}, o bien usar el constructor de la clase `set()` y pasarle como argumento un objeto *iterable* (como una *lista*, una *tupla*, una *cadena* ⋯).

```
# Crea un conjunto con una serie de elementos entre llaves #
Los elementos repetidos se eliminan
c = {1, 3, 2, 9, 3, 1}
c
{1, 2, 3, 9}
# Crea un conjunto a partir de un string
# Los caracteres repetidos se eliminan a
= set('Hola Python')
a
{'a', 'H', 'h', 'y', 'n', 's', 'P', 't', ' ', 'i', 'l', 'o'}
# Crea un conjunto a partir de una lista
# Los elementos repetidos de la lista se eliminan
unicos = set([3, 5, 6, 1, 5])
unicos
```

{1, 3, 5, 6}

Para crear un conjunto vacío, simplemente llama al constructor `set()` sin parámetros.

**set vs frozenset**

En realidad, en Python existen dos clases para representar conjuntos: `set` y `frozenset`. La principal diferencia es que *set* es mutable, por lo que después de ser creado, se pueden añadir y/o eliminar elementos del conjunto, como veremos en secciones posteriores. Por su parte, *frozenset* es inmutable y su contenido no puede ser modificado una vez que ha sido inicializado.

Para crear un conjunto de tipo *frozenset*, se usa el constructor de la clase `frozenset()`: f

```
= frozenset([3, 5, 6, 1, 5])
f
frozenset({1, 3, 5, 6})
```

Dado que los conjuntos son colecciones desordenadas, en ellos no se guarda la posición en la que son insertados los elementos como ocurre en los tipos list o tuple. Es por ello que no se puede acceder a los elementos a través de un índice.

Sin embargo, sí se puede acceder y/o recorrer todos los elementos de un conjunto usando un bucle for:

```
mi_conjunto = {1, 3, 2, 9, 3, 1}
for e in mi_conjunto:
... print(e)
...
1
2
3
9
```

Para añadir un elemento a un conjunto se utiliza el método `add()`. También existe el método `update()`, que puede tomar como argumento una *lista*, *tupla*, *string*, *conjunto* o cualquier objeto de tipo *iterable*.

```
mi_conjunto = {1, 3, 2, 9, 3, 1}
mi_conjunto
{1, 2, 3, 9}
# Añade el elemento 7 al conjunto
mi_conjunto.add(7)
mi_conjunto
{1, 2, 3, 7, 9}
# Añade los elementos 5, 3, 4 y 6 al conjunto
# Los elementos repetidos no se añaden al conjunto
mi_conjunto.update([5, 3, 4, 6])
mi_conjunto
{1, 2, 3, 4, 5, 6, 7, 9}
```

**NOTA:** `add()` y `update()` no añaden elementos que ya existen al conjunto

La clase *set* ofrece cuatro métodos para eliminar elementos de un conjunto. Son: `discard()`, `remove()`, `pop()` y `clear()`.

`discard(elemento)` y `remove(elemento)` eliminan `elemento` del conjunto. La única diferencia es que lanza si `elemento` no existe, `discard()` no hace nada mientras que `remove()` la excepción KeyError.

`pop()` Este método devuelve un elemento aleatorio del conjunto y lo elimina del mismo. Si el conjunto está vacío, lanza la excepción `KeyError`.

`clear()` elimina todos los elementos contenidos en el conjunto.

```
mi_conjunto = {1, 3, 2, 9, 3, 1, 6, 4, 5}
```

```
mi_conjunto
{1, 2, 3, 4, 5, 6, 9}
# Elimina el elemento 1 con remove()
mi_conjunto.remove(1)
mi_conjunto
{2, 3, 4, 5, 6, 9}
# Elimina el elemento 4 con discard()
mi_conjunto.discard(4)
mi_conjunto
{2, 3, 5, 6, 9}
# Trata de eliminar el elemento 7 (no existe) con remove() #
Lanza la excepción KeyError
mi_conjunto.remove(7)
Traceback (most recent call last):
File "<input>", line 1, in <module>
KeyError: 7
# Trata de eliminar el elemento 7 (no existe) con discard() #
No hace nada
mi_conjunto.discard(7)
mi_conjunto
{2, 3, 5, 6, 9}
# Obtiene y elimina un elemento aleatorio con pop() mi_conjunto.pop()
2
mi_conjunto
{3, 5, 6, 9}
# Elimina todos los elementos del conjunto
mi_conjunto.clear()
mi_conjunto
set()
```

Como con cualquier otra colección, puedes usar la función `len()` para obtener el número de elementos contenidos en un conjunto:

```
mi_conjunto = set([1, 2, 5, 3, 1, 5])
len(mi_conjunto)
4
```

Con los conjuntos también se puede usar el operador de pertenencia `in` para comprobar si un elemento está contenido, o no, en un conjunto:

```
mi_conjunto = set([1, 2, 5, 3, 1, 5])
print(1 in mi_conjunto)
True
print(6 in mi_conjunto)
False
print(2 not in mi_conjunto)
False
```

Los principales usos del tipo `set` es utilizarlo en operaciones del álgebra de conjuntos: *unión*, *intersección*, *diferencia*, *diferencia simétrica*, ···

**Unión de conjuntos en Python**

La unión de dos conjuntos A y B es el conjunto A U B que contiene todos los elementos de A y de B.

En Python se utiliza el operador | para realizar la unión de dos o más conjuntos. a
= {1, 2, 3, 4}
b = {2, 4, 6, 8}
a | b
{1, 2, 3, 4, 6, 8}

**Intersección de conjuntos en Python**

La intersección de dos conjuntos A y B es el conjunto A ∩ B que contiene todos los elementos comunes de A y B.

En Python se utiliza el operador & para realizar la intersección de dos o más conjuntos.
a = {1, 2, 3, 4}
b = {2, 4, 6, 8}
a & b
{2, 4}

**Diferencia de conjuntos en Python**

La diferencia entre dos conjuntos A y B es el conjunto A \ B que contiene todos los elementos de A que no pertenecen a B.

En Python se utiliza el operador - para realizar la diferencia de dos o más conjuntos. a
= {1, 2, 3, 4}
b = {2, 4, 6, 8}
a - b
{1, 3}

**Diferencia simétrica de conjuntos en Python**

La diferencia simétrica entre dos conjuntos A y B es el conjunto que contiene los elementos de A y B que no son comunes.

En Python se utiliza el operador ^ para realizar la diferencia simétrica de dos o más conjuntos. a
= {1, 2, 3, 4}
b = {2, 4, 6, 8}
a ^ b
{1, 3, 6, 8}

Los métodos principales de la clase *set* en Python:

| Método | Descripción |
|---|---|
| add(e) | Añade un elemento al conjunto. |

| Método | Descripción |
|---|---|
| clear() | Elimina todos los elementos del conjunto. |
| copy() | Devuelve una copia superficial del conjunto. |
| difference(iterable) | Devuelve la diferencia del conjunto con el iterable como un conjunto nuevo. |
| difference_update(iterable) | Actualiza el conjunto tras realizar la diferencia conel iterable. |
| discard(e) | Elimina, si existe, el elemento del conjunto. |
| intersection(iterable) | Devuelve la intersección del conjunto con el iterable como un conjunto nuevo. |
| intersection_update(iterable) | Actualiza el conjunto tras realizar la intersección conel iterable. |
| isdisjoint(iterable) | Devuelve True si dos conjuntos son disjuntos. |
| issubset(iterable) | Devuelve True si el conjunto es subconjunto del iterable. |
| issuperset(iterable) | Devuelve True si el conjunto es superconjunto del iterable. |
| pop() | Obtiene y elimina un elemento de forma aleatoria del conjunto. |
| remove(e) | Elimina el elemento del conjunto. Si no existe lanza un error. |
| symmetric_difference(iterable) | Devuelve la diferencia simétrica del conjunto conel iterable como un conjunto nuevo. |
| symmetric_difference_update(iterable) | Actualiza el conjunto tras realizar la diferencia simétrica con el iterable. |
| union(iterable) | Devuelve la unión del conjunto con el iterable como un conjunto nuevo. |

| Método | Descripción |
| --- | --- |
| update(iterable) | Actualiza el conjunto tras realizar la unión con el iterable. |

**Diccionarios (dict) en Python**

La clase dict de Python es un tipo mapa que asocia claves a valores. A diferencia de los tipos secuenciales (*list*, *tuple*, *range* o *str*), que son indexados por un índice numérico, los diccionarios son indexados por claves. Estas claves siempre deben ser de un tipo inmutable.

Piensa siempre en un diccionario como un contenedor de pares *clave*: *valor*, en el que la clave puede ser de cualquier tipo 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 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 loharíamos con otros tipos de datos.

Otras características para resaltar de los diccionarios:

- **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*.

En Python hay varias formas de crear un diccionario. Las veremos todas a continuación.

La más simple es encerrar una secuencia de pares *clave: valor* separados por comas entre llavesd {}
= {1: 'hola', 89: 'Pythonista', 'a': 'b', 'c': 27}

En el diccionario anterior, los enteros 1 y 89 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*es 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:

```
# 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
{}
```

El acceso a un valor se realiza mediante indexación de la clave. Para ello, simplemente encierra entre corchetes la clave del elemento 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 si no encuentra la clave
d.get('cuatro', 4)
4
# Devuelve None como valor por defecto si no encuentra la clave a
= d.get('cuatro')
```

```
a
type(a)
<class 'NoneType'>
Cómo usar el bucle for para recorrer un diccionario. d
= {'uno': 1, 'dos': 2, 'tres': 3}
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)
```

La clase *dict* es mutable, por lo que se pueden añadir, modificar y/o eliminar elementos después de haber 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}
```

Para actualizar el valor asociado a una clave, simplemente se asigna 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}
```

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
{}
```

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'
```

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()`. Esto devolverá las claves del diccionario en una lista.

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

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:

- `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.

Los principales métodos de la clase *dict*. Algunos de ellos ya los hemos visto en las secciones anteriores:

| Método | Descripción |
|---|---|
| `clear()` | Elimina todos los elementos del diccionario. |
| `copy()` | Devuelve una copia poco profunda del diccionario. |
| `get(clave[, valor])` | Devuelve el valor de la `clave`. Si no existe, devuelve el valor `valor` si se indica y si no, `None`. |

| Método | Descripción |
| --- | --- |
| `items()` | Devuelve una vista de los pares *clave: valor* del diccionario. |
| `keys()` | Devuelve una vista de las claves del diccionario. |
| `pop(clave[, valor])` | Devuelve el valor del elemento cuya clave es `clave` y elimina el elemento del diccionario. Si la clave no se encuentra, devuelve `valor` si se proporciona. Si la clave no se encuentra y no se indica `valor`, lanza la excepción `KeyError`. |
| `popitem()` | Devuelve un par *(clave, valor)* aleatorio del diccionario. Si el diccionario está vacío, lanza la excepción `KeyError`. |
| `setdefault(clave[, valor])` | Si la `clave` está en el diccionario, devuelve su valor. Si no lo está, inserta la `clave` con el valor `valor` y lo devuelve (si no se especifica `valor`, por defecto es `None`). |
| `update(iterable)` | Actualiza el diccionario con los pares *clave: valor* del iterable. |
| `values()` | Devuelve una vista de los valores del diccionario. |