



Hoja informativa: Tipos de datos

Tipos de datos

Los tipos de datos básicos de Python son los siguientes:

- Enteros: `int`.
- Números reales: `float`.
- Strings: `str`.
- Tipo lógico: `bool`.

Estos se utilizan muy seguido y se encuentran en casi todos los programas.

Todos estos tipos de datos son **integrados** e **inmutables**.

Para utilizarlos no es necesario descargar otras librerías ni introducir configuraciones adicionales, el intérprete de Python ya los reconoce.

Además, no puedes editar sus valores, solo sobrescribirlos.

Cada tipo de datos tiene sus propios usos.

Números enteros y reales

Si necesitas calcular algo, utiliza números enteros o reales. Los números reales son números con una parte fraccionaria, por ejemplo, `1.12`.



La parte decimal de la fracción en el código está separada por un punto; no utilices comas en este caso porque Python no las comprende.

Por ejemplo, puedes multiplicar π por 2:

```
pi = 3.1415926535
result = pi * 2

print(result)          # mostrará: 6.283185307
```

Las variables `pi` y `result` almacenan números reales: `float`. Pero `2` es un número entero.

Strings

Los strings se utilizan para trabajar con textos o conjuntos de caracteres. En el código, se comportan como texto normal.

Los strings siempre se encierran entre comillas simples o dobles, puedes decidir qué formato utilizar, pero asegúrate de utilizar comillas uniformes en todo el código.

```
title = 'Matrix'           # Un string entre comillas sencillas
directors = "Lilly y Lana Wachowski" # Un string entre comillas dobles
year = "1999"              # Con comillas, los números también son strings
```

Las comillas son parte de la sintaxis de Python: las utiliza para identificar un string.

Tipo booleano

Este es el tipo de datos `bool`. Puede tomar uno de estos dos valores:

- `True`: verdadero en inglés
- `False`: falso en inglés



Estos dos valores se escriben con una letra mayúscula y sin comillas.

Estas son palabras clave reservadas en Python. Por lo tanto, no se pueden utilizar como nombres de variables.

Necesitas el tipo lógico para escribir expresiones lógicas.

Listas

Las listas son colecciones ordenadas de elementos; consisten de elementos numerados.

El número de un elemento de la lista se llama índice. Puedes utilizar el índice para acceder al elemento de la lista correspondiente.

Listas: uso

Las listas nos permiten almacenar múltiples valores en una sola variable.

Por lo general, las listas se utilizan para almacenar valores del mismo tipo de datos. Digamos que tienes botones que se deben probar. Sin utilizar una lista, tendrías que crear variables separadas para cada nombre de botón:

```
# Debes probar tres botones
# Si guardas cada uno en una variable separada, tendrás mucho código:
button_1 = "registration_button"
button_2 = "enter_button"
button_3 = "help_button"
```

Puedes optimizar esto mediante una lista:

```
# Crea una nueva variable llamada buttons (botones)
# Asignala a una lista de tres botones
```

```
buttons = ["registration_button", "enter_button", "help_button"]
```

Las listas son modificables o mutables. Esto significa que no necesitas saber previamente cuántos elementos acomodará la lista.

Para crear una lista, haz lo siguiente:

1. Especifica el nombre de la lista. Es el mismo que el nombre de la variable donde se almacenará la lista. Por ejemplo, `buttons`.
2. Utiliza el operador de asignación `=`.
3. Utiliza corchetes: `[` y `]`. Python los utiliza para identificar una lista.
4. Dentro de los corchetes, enumera los elementos separados por comas. Por ejemplo, `"registration_button", "enter_button" y "help_button"`.

Es necesario utilizar comas, sin ellas, Python no podrá determinar dónde termina un elemento y comienza otro.

```
buttons = ["registration_button", "enter_button", "help_button"]
```

Indexación


Todos los elementos de una lista siguen un orden estricto.

```
# Por ejemplo, some_list guarda nueve letras
some_list = ["b", "r", "i", "l", "l", "i", "a", "n", "t"]
```

Cada elemento en esta lista tiene su propio número de **índice**.



La indexación en las listas comienza con `0`, por lo que el primer elemento siempre está en el índice 0.


`some_list = ["b", "r", "i", "l", "l", "i", "a", "n", "t"]`

Cómo acceder y reasignar elementos de la lista

Para trabajar con un solo elemento de la lista, puedes **acceder** a él.

Debes hacer lo siguiente:

1. Especifica el nombre de la lista.
2. Utiliza corchetes.
3. Escribe el índice del elemento entre corchetes.

Por ejemplo, el siguiente código mostrará "i" desde `some_list`:

```
print(some_list[2]) # Salida: "i"
```

`some_list = ["b", "r", "i", "l", "l", "i", "a", "n", "t"]`

Nombre de la lista

Índice del elemento entre corchetes

Métodos de las listas

Las listas son modificables (mutables), puedes agregar elementos utilizando el método `append()`:

```
# Agrega "!" al final de some_list
some_list.append("!")

print(some_list) # Salida: ["b", "r", "i", "l", "l", "i", "a", "n", "t", "!"]
```

Puedes eliminar elementos mediante `pop()`:

```
# Elimina el elemento en el índice 0
# Ahora, es la letra 'b'
some_list.pop(0)
print(some_list) # Salida: ["r", "i", "l", "l", "i", "a", "n", "t", "!"]

# Después de eliminar un elemento, la indexación cambia:
print(some_list[0]) # Salida: r
```

Puedes ordenar los elementos de la lista mediante `sort()`:

```
# Ordena los elementos de la lista en orden ascendente
some_list.sort()
```

```
print(some_list) # Salida: ['!', 'a', 'i', 'i', 'l', 'l', 'n', 'b', 'r', 't']
```

Para utilizar estos métodos, haz lo siguiente:

1. Especifica el nombre de la lista. En nuestro caso, `some_list`.
2. Coloca un punto.
3. Especifica el método. Por ejemplo, `append()`.
4. Pasa los datos requeridos entre paréntesis: el elemento que deseas agregar o el índice del elemento que deseas eliminar. Al ordenar listas, los paréntesis se dejan vacíos.



Cada vez que elimines u ordenes los elementos de la lista, realiza un seguimiento de la indexación a medida que los elementos cambian de posición.

Bucles

¿Qué son los bucles?

Las pruebas a menudo requieren realizar una serie de acciones idénticas, por ejemplo, iterando sobre todas las filas de la tabla en una base de datos. Este proceso no es infinito: continúa solo **con la condición** de que falten filas de tabla por procesar.

Ejemplo. Una sala de cine está organizando un festival de cine de culto. Tienes una lista de películas de 1994:

```
best_movies = ['Sueños de fuga', 'Tiempos violentos', 'Forrest Gump', 'El perfecto asesino', 'El rey león']
```

Debes mostrar los títulos en la cartelera:

```
Sueños de fuga
Tiempos violentos
Forrest Gump
El perfecto asesino
El rey león
```

Podrías mostrar cada título de forma manual, pero eso es ineficaz y requiere mucho tiempo:

```
print(best_movies[0])
print(best_movies[1])
...
```

Para resolver la tarea, debes hacer lo siguiente: toma el primer elemento de `best_movies`, muéstralo mediante la función `print()`, toma el siguiente elemento y continúa hasta el final de la lista.

Para este tipo de tareas, el equipo de desarrollo ha inventado los **loops** o **bucles**, estos son fragmentos de código especiales que repiten acciones particulares mientras una determinada condición sea verdadera.

Cómo escribir un bucle for

Un bucle para nuestra cartelera tiene este aspecto:

```
best_movies = ['Sueños de fuga', 'Tiempos violentos', 'Forrest Gump', 'El perfecto asesino', 'El rey león']

for movie in best_movies: # Esta es la condición del bucle
    print(movie) # Y este es el cuerpo del bucle
```

Observa las dos líneas de código después de la declaración de la lista `best_movies`. Este es un bucle for, tiene dos partes: una **definición** y un **cuerpo**.

Definición del bucle

Primero, debes ordenarle al programa que inicie un bucle for. En Python, los bucles for se declaran con las palabras clave `for` e `in`. Estas palabras clave son parte de la **definición del bucle**, que termina con dos puntos.

```
for <variable> in <list>: # Sintaxis general
```

Para nuestras películas, se ve así:

```
for movie in best_movies:
```

La definición del bucle contiene lo siguiente:

- `for` está seguido por el nombre de la variable que recibirá los elementos de la lista uno por uno. Esta variable se llama **variable del bucle**.
- `in` está seguido por el nombre de la lista que se va a procesar.

Variable de bucle

Una variable de bucle es una especie de contenedor para el elemento que necesita ser procesado por el bucle.

Una vez procesado el primer elemento de la lista, se asigna a la variable el siguiente elemento, continuando este proceso hasta el final de la lista.

Observa que la variable `movie` apareció primero en la definición de bucle. No se ha declarado antes en ninguna parte del código. La variable recibe primero un nombre en la definición del bucle.



Puedes nombrar la variable de bucle de la forma que desees, pero es habitual nombrarla igual que la lista, en forma singular.

Si el nombre de la lista es `musicians` (músicos), la variable se llama `musician`. Si el nombre de la lista es `friends`, la variable se llama `friend`. Esta regla hace que sea más fácil leer y escribir el código.

Siguiendo la definición del bucle, el **cuerpo del bucle** comienza en una nueva línea.

Cuerpo del bucle

Este es el bloque de código que se ejecutará en cada iteración del bucle, es decir, para cada elemento de la lista.

La definición del bucle le dice al programa **qué** se supone que debe procesar el bucle, y el cuerpo especifica **cómo**.

Cada línea del cuerpo del bucle debe tener una indentación de cuatro espacios:

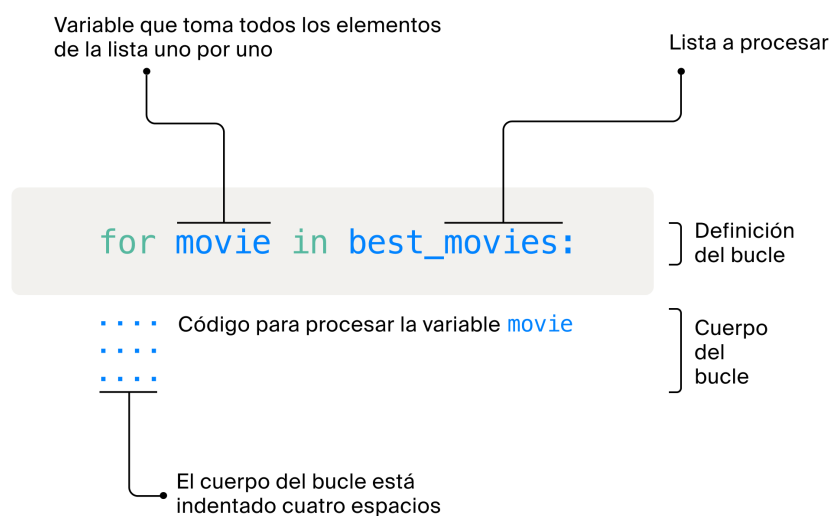
```
for <variable> in <list>:
    # El cuerpo del bucle: este bloque de código se ejecutará para cada elemento
    # Aquí, puedes hacer algo con la variable declarada en la condición del bucle

for movie in best_movies:
    print(movie) # Por ejemplo,
                # muéstrala
```

Python utiliza indentaciones para determinar dónde comienza y termina el cuerpo del bucle.

! La indentación es una regla técnica estricta dentro de Python. Sin ella, el programa no se ejecutará.

Cómo funcionan los bucles for



El bucle toma el primer elemento de la lista `best_movies` y lo pasa a la variable `movie`. Luego, ejecuta el bloque de código en el cuerpo del bucle, mostrando el valor de la variable `movie`.

El elemento que se mostrará primero es `Sueños de fuga`. Entonces, comienza una nueva iteración. La variable del bucle recibe `Tiempos violentos`. El bucle muestra su valor.

Esto continúa hasta el final de la lista:

```
Sueños de fuga # El resultado de la primera iteración
Tiempos violentos # El resultado de la segunda iteración
Forrest Gump # Y así sucesivamente
El perfecto asesino
El rey león
```

Cada ciclo se conoce como una **iteración de bucle**.

Cuando la lista termina, el bucle se cierra y Python avanza a la siguiente línea de código.

Bucles con condiciones

Algunas veces debes iterar sobre todos los elementos de una secuencia, pero procesar sólo algunos de ellos. Digamos que necesitas encontrar un valor determinado en la base de datos y mostrarlo.

En este caso, debes iterar a través de todos los elementos y encontrar aquellos que satisfagan tus criterios, por ejemplo, todos los números pares.

Tareas como esta se pueden resolver mediante la ramificación de código con sentencias `if...else`. Estas sentencias se pueden utilizar dentro de un bucle.

Ejemplo. Digamos que tu bolso está en el casillero número 3. Vamos a declarar una lista llamada `lockers` (casilleros) con los números de casilleros e iterar a través de ellos para encontrar el número de tu casillero.

```
lockers = [1, 2, 3, 4, 5]

for locker in lockers:
    # Cuerpo del bucle
    if locker == 3: # Sentencia if...else anidada
        print('¡Aquí está mi equipaje!') # El programa muestra este mensaje al encontrar el número del casillero
```

El bucle itera sobre todos los números de casillero, comparando el número actual con el número requerido. Cuando el bucle llega al casillero número 3, muestra el mensaje.

Sentencias `break` y `continue` en bucles

Cuando hay una sentencia `if` en el bucle, el programa busca los valores correctos para procesar, por ejemplo, para mostrarlos. Para hacerlo, el programa itera a través de la secuencia completa, elemento por elemento.

Hay un problema, el programa continúa ejecutándose incluso después de haber localizado los valores requeridos.

Ejemplo. Imagina que estás buscando el casillero donde está tu bolso, pero no sabes su número. Necesitas iterar a través de todos los casilleros en el almacenamiento. Hay 999 en total.

Tu equipaje todavía está en el casillero número 3. Cuando encuentras el número 3, recuerdas que ahí está, pero aún no lo abres y sigues comprobando todos los casilleros hasta que ya no quede ninguno.

Así es exactamente como suele funcionar un bucle:

```
lockers = range(1, 1000) # Números de casillero

for i in lockers:
    if i == 3: # Sentencia if anidada
        print('¡Aquí está mi equipaje!') # Cuando se cumple la condición,
                                         # se muestra este mensaje

    print('Comprobando', str(i), 'casillero')
    # Esta línea muestra todos los valores de la variable i
    # El código procesará el rango completo en cualquier caso
    # Puedes ver cuánto trabajo redundante hace
```

Salida:

```
Comprobando casillero 1
Comprobando casillero 2
¡Aquí está mi equipaje!
Comprobando casillero 3
Comprobando casillero 4
Comprobando casillero 5
...
<Aquí, el bucle itera de 5 a 996>
...
Comprobando casillero 997
Comprobando casillero 998
Comprobando casillero 999
```

Python tiene una herramienta que permite optimizar este bucle: las **palabras clave** `continue` y `break`. Con estas, el bucle no procesará ningún elemento innecesario.

Omisión de iteración de bucle: `continue`

La palabra clave `continue` se utiliza en conjunto con una sentencia `if` anidada para omitir una iteración.

Ejemplo. Digamos que tenemos una lista llamada `ovejas` con números del 1 al 5. Debemos utilizar un bucle para mostrar todos los valores excepto el tercero.

De esta forma:

- Escribe una sentencia anidada `if i == 3`
- Agrega una sentencia `continue` **dentro** de la sentencia

```
sheep = [1, 2, 3, 4, 5]

for i in sheep: # Bucle exterior
    if i == 1:
```

```

        print(i, ' oveja')
    else:
        print(i, ' ovejas')
    if i == 3: # Sentencia if anidada: si la variable es igual a 3,
        continue # omite la iteración

print('¿Adónde se fue una oveja?') # Esta línea está fuera del bucle

```

El programa mostrará esto:

```

1 oveja
2 ovejas
4 ovejas
5 ovejas
¿A dónde se fue una oveja?

```

Explicación. Una vez que la variable `i` recibió `3`, se cumplió la condición `if i == 3`. La sentencia `continue` le ordenó al bucle que omitiera esta iteración.

Interrupción de bucle: `break`

La sentencia `break` interrumpe la ejecución del bucle.

Ejemplo. La variable `new_sheep` contiene números del 1 al 10. Debes mostrar los primeros cuatro valores utilizando un bucle y luego interrumpir el bucle.

Para hacerlo:

- Escribe una sentencia anidada `if i == 5:`
- Agrega una sentencia `break` dentro de la sentencia `if`

```

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

for i in new_sheep:
    if i == 1:
        print(i, ' oveja')
    else:
        print(i, ' ovejas')
    if i == 5:
        break

print('¿Dónde están todas las demás?')

```

La salida:

```

1 oveja
2 ovejas
3 ovejas
4 ovejas
¿Dónde están todas las demás?

```

Explicación. Una vez que la variable `i` recibió `5`, se cumplió la condición `if i == 5`. La sentencia `break` ordenó que se cerrara el ciclo.



El uso de estas sentencias evita que el programa realice un trabajo innecesario y te ahorra tiempo.

Tuplas

Una `tupla` de Python es una colección inmutable de elementos. Así es como se ve:

```
some_tuple = ('p', 'y', 't', 'h', 'o', 'n') # Esta es una tupla
# Los elementos de la tupla están entre paréntesis
```

Cómo declarar una tupla

Para crear una tupla, haz lo siguiente:

1. Escribe el nombre de la tupla seguido del operador de asignación.
2. Utiliza paréntesis: `(` and `)`. Python los utiliza para identificar una tupla.
3. Dentro de los paréntesis, enumera los elementos separados por comas.

Las tuplas y las listas se comportan de manera diferente en el código.



No puedes sobrescribir, agregar, eliminar u ordenar elementos de tupla.

Las tuplas se pueden utilizar como un todo en el programa:

```
some_tuple = ('p', 'y', 't', 'h', 'o', 'n')

print(some_tuple) # Salida: ('p', 'y', 't', 'h', 'o', 'n')
```

También es posible procesar elementos de tupla por separado.

Al igual que con las listas, para utilizar un elemento de tupla, debes acceder a este por índice:

```
some_tuple = ('p', 'y', 't', 'h', 'o', 'n')

# Muestra el quinto elemento de la tupla
print(some_tuple[4]) # Salida: o
```

Cuándo utilizar tuplas

Las tuplas se utilizan para almacenar datos que deben permanecer sin cambios a lo largo del programa.

Lista

- Los elementos y su orden se pueden cambiar
- Como los nombres de botones, películas favoritas o comestibles en la nevera

Tupla

- Los elementos y su orden son fijos
- Como las estaciones o los días de la semana

Diccionarios

Los diccionarios son colecciones de pares clave-valor.

Cada elemento del diccionario consta de dos partes: la primera parte es una **clave** y la segunda es un **valor**. Y se separan con dos puntos.

El diccionario `user_213` almacena datos del usuario o usuaria: nombre, edad e idiomas.

```
# Diccionario con datos del usuario o usuaria
user_213 = {"first_name": "Linda",          # Primer par
            "age": 26,                      # Segundo par
            "languages": ["Inglés", "Español"]} # Tercer par
```

Elementos de sintaxis requeridos: llaves, dos puntos, comas

```
user_213 = {
    'first_name': 'Linda',
    'age': '26',
    'languages': ['Inglés', 'Español']
}
```

Claves

Valores

Al crear claves de diccionario debes considerar ciertas reglas.

Reglas para claves de diccionario

✓ Solo los tipos de datos inmutables pueden ser **claves** de diccionario. Estos incluyen lo siguiente:

- Números

- Strings
- Tipo booleano `True` y `False`

✗ Los diccionarios **no permiten duplicar las claves**. Una clave es una referencia única utilizada por Python para encontrar el valor correspondiente en el programa. Por eso no puede haber claves duplicadas.

Si agregas dos claves idénticas a un diccionario, perderás uno de los valores. La clave se agregará al diccionario una sola vez y asumirá el último valor introducido.

Reglas para valores de diccionario

✓ Todos los tipos de datos pueden ser **valores** de diccionario: números, strings, listas e incluso otros diccionarios.

Vamos a convertir nuestra lista `bag` en un diccionario. Las claves serán los compartimentos, y los valores serán el contenido.

```
bag = {"compartimento principal":["laptop", "cargador", "agua"], # Esto es lo que obtenemos
      "bolsillo frontal":["llaves", "cartera"],
      "bolsillo lateral":["un trozo de chocolate"]}

print(bag)
```

✓ Se permiten valores duplicados. La variable `button` contiene un diccionario con botones. Las claves son los nombres de los botones y los valores son sus colores. Los botones **"Log in"** (Iniciar sesión) y **"Sign up"** (Registrarse) son de color negro.

```
buttons = {"registration_button":"negro", # Esto es lo que obtenemos
          "enter_button":"negro",        # El mismo valor "negro" se utiliza dos veces,
          "help_button":"rojo"}          # pero no causará un error

print(buttons)
```

Uso de diccionarios

Las listas y las tuplas son excelentes para almacenar datos del mismo tipo, se utilizan cuando todos los valores de una variable están en la misma categoría, por ejemplo, nombres de botones o elementos de menú.

```
# Carta de restaurante como lista
menu = ["tortilla", "café", "tostada", "ratatouille", "sopa de champiñones", "té"]
```

Pero algunas veces necesitas almacenar valores en diferentes grupos, por ejemplo, una carta de desayunos separada y una carta de comida de negocios separada.

En este caso, es mejor utilizar un diccionario. Te permite dividir valores por claves mientras se almacenan en el mismo lugar, lo que facilita encontrarlos y reutilizarlos.

```
# Carta de restaurante como diccionario
# Hay dos claves: "desayuno" y "comida de negocios"

menu = {"desayuno":["tortilla", "café", "tostada"]
        "comida de negocios":["ratatouille", "sopa de champiñones", "té"]}
```

Cómo acceder a un valor

Para acceder a un valor de diccionario, debes consultar su clave. Esto también se conoce como **acceder a un valor mediante una clave**.

El acceso mediante una clave es similar al acceso mediante un índice en las listas: debes especificar el nombre del diccionario seguido de la clave requerida entre corchetes.

Así es como puedes recuperar la edad del usuario o usuaria del diccionario `user_213`:

```
user_213 = {"first_name":"Linda",
            "age":26,
            "languages":["Inglés", "Español"]}

print(user_213["edad"]) # Accede a la edad del usuario o usuaria
```

Operaciones de diccionario

Añadir elementos

Para añadir un nuevo elemento a un diccionario, sigue estos pasos:

1. Especifica el nombre del diccionario seguido de corchetes. por ejemplo, `dict_name[]`.
2. En los corchetes, especifica la nueva clave del diccionario, por ejemplo, `new_key`.
3. Utiliza el operador de asignación para especificar el valor que deseas añadir, por ejemplo, `= new_value`.

Esto es lo que obtenemos:

```
# Agrega el par new_key=new_value a dict_name
dict_name[new_key] = new_value
```



Nota: la clave debe ser nueva.

De lo contrario, modificarás el elemento existente en lugar de agregar uno nuevo. Después del almuerzo, hay un artículo nuevo en tu bolso: una caja de jugo. Así es como se ve en el código:

```
# Declara un diccionario llamado bag
bag = {"compartimento principal":["laptop", "cargador", "agua"],
       "bolsillo frontal":["llaves", "cartera"],
```

```
"bolsillo lateral":["un trozo de chocolate"]}  
  
bag["bolsillo lateral derecho"] = ["una caja de jugo"] # Se agregó un artículo nuevo al bolso  
# "bolsillo lateral derecho" es la clave, y "caja de jugo" es un valor
```

Sobrescribir elementos

Cambiar el valor de un elemento es similar a agregar uno nuevo, la única diferencia es que debes especificar la clave existente en lugar de una nueva.

1. Especifica el nombre del diccionario, luego agrega corchetes después de él.
2. Entre corchetes, especifica la clave para el valor que deseas cambiar.
3. Coloca un operador de asignación para especificar el nuevo valor.

```
# Reasigna el valor de la clave como key_value en el diccionario dict_name  
dict_name[key] = new_value
```

Digamos que necesitas reemplazar la caja de jugo por un refresco en el diccionario `bag`:

```
bag = {"compartimento principal":["laptop", "cargador", "agua"],  
      "bolsillo frontal":["llaves", "cartera"],  
      "bolsillo lateral izquierdo":["un trozo de chocolate"],  
      "bolsillo lateral derecho":["una caja de jugo"]}  
  
bag["bolsillo lateral derecho"] = ["un refresco"]  
  
print(bag)
```

Eliminar elementos

Puedes cambiar los valores del diccionario, pero no las claves del diccionario. Si deseas cambiar una clave, debes eliminar el elemento y agregar uno nuevo: el mismo valor, pero con una clave diferente.

Para eliminar un elemento del diccionario, utiliza el método `pop()`. El método funciona con los diccionarios de la misma manera que con las listas, solo que debes especificar la clave del diccionario en lugar del índice.

```
# Elimina un elemento con clave, como el nombre de su clave, de dict_name  
dict_name.pop(key)
```

Por ejemplo, debes eliminar el elemento con la clave `"bolsillo lateral izquierdo"` del diccionario `bag`:

```
bag.pop("bolsillo lateral izquierdo")
```

Operadores de pertenencia

En ocasiones, necesitas verificar si un valor particular está presente en una colección de datos. En este caso, puedes utilizar operadores de pertenencia. Estos te permiten validar la pertenencia de un valor en una secuencia.

Solo hay dos operadores de pertenencia:

Función	Operador	Ejemplo
Comprueba que una secuencia contiene un valor	<code>in</code>	<code>"d" in ("a", "d")</code>
Comprueba que una secuencia no contiene un valor	<code>not in</code>	<code>"z" not in ("a", "d")</code>

Cómo utilizar estos operadores:

1. Escribe el nombre del elemento que buscas. En nuestro caso, `"miel"`.
2. Escribe el operador de pertenencia.
3. Escribe el nombre de la variable que almacena la colección de datos. En nuestro caso, `recipe`.

Esto es lo que obtenemos: `"miel" en recipe`.

```
recipe = ["leche", "mantequilla", "huevos", "azúcar", "harina", "azúcar", "vainilla", "nueces", "miel"]  
  
print("miel" in recipe) # Salida: True
```

El resultado es un tipo de datos booleano: `True` o `False`.

Los operadores de pertenencia se pueden utilizar con listas, tuplas, strings y diccionarios.

Esto es lo que prueban:

- En strings: caracteres y conjuntos de caracteres.
- En listas y tuplas: elementos.
- En diccionarios: solo claves.