

## CAPÍTULO 5

---

### Estructuras de datos

---

Si bien ya hemos visto una sección sobre *Tipos de datos*, podríamos hablar de tipos de datos más complejos en Python que se constituyen en **estructuras de datos**. Si pensamos en estos elementos como *átomos*, las estructuras de datos que vamos a ver sería *moléculas*. Es decir, combinamos los tipos básicos de formas más complejas. De hecho, esta distinción se hace en el [Tutorial oficial de Python](#). Trataremos distintas estructuras de datos como listas, tuplas, diccionarios y conjuntos.

## 5.1 Listas



Las listas permiten **almacenar objetos** mediante un **orden definido** y con posibilidad de duplicados. Las listas son estructuras de datos **mutables**, lo que significa que podemos añadir, eliminar o modificar sus elementos.<sup>1</sup>

### 5.1.1 Creando listas

Una lista está compuesta por *cero o más elementos*. En Python debemos escribir estos elementos separados por *comas* y dentro de *corchetes*. Veamos algunos ejemplos de listas:

```
>>> empty_list = []

>>> languages = ['Python', 'Ruby', 'Javascript']

>>> fibonacci = [0, 1, 1, 2, 3, 5, 8, 13]

>>> data = ['Tenerife', {'cielo': 'limpio', 'temp': 24}, 3718, (28.2933947, -16.
↪5226597)]
```

---

**Nota:** Una lista puede contener tipos de **datos heterogéneos**, lo que la hace una estructura

---

<sup>1</sup> Foto original de portada por [Mike Arney](#) en Unsplash.

de datos muy versátil.

---

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/Ofiiare>

## 5.1.2 Conversión

Para convertir otros tipos de datos en una lista podemos usar la función `list()`:

```
>>> # conversión desde una cadena de texto
>>> list('Python')
['P', 'y', 't', 'h', 'o', 'n']
```

Si nos fijamos en lo que ha pasado, al convertir la cadena de texto `Python` se ha creado una lista con 6 elementos, donde cada uno de ellos representa un carácter de la cadena. Podemos *extender* este comportamiento a cualquier otro tipo de datos que permita ser iterado (*iterables*).

### Lista vacía

Existe una manera particular de usar `list()` y es no pasarle ningún argumento. En este caso estaremos queriendo convertir el «vacío» en una lista, con lo que obtendremos una *lista vacía*:

```
>>> list()
[]
```

---

**Truco:** Para crear una lista vacía, se suele recomendar el uso de `[]` frente a `list()`, no sólo por ser más *pitónico* sino por tener (en promedio) un mejor rendimiento en tiempos de ejecución.

---

## 5.1.3 Operaciones con listas

### Obtener un elemento

Igual que en el caso de las *cadena de texto*, podemos obtener un elemento de una lista a través del **índice** (lugar) que ocupa. Veamos un ejemplo:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping[0]
'Agua'

>>> shopping[1]
'Huevos'

>>> shopping[2]
'Aceite'

>>> shopping[-1] # acceso con índice negativo
'Aceite'
```

El *índice* que usamos para acceder a los elementos de una lista tiene que estar comprendido entre los límites de la misma. Si usamos un índice antes del comienzo o después del final obtendremos un error (*excepción*):

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

>>> shopping[-5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

### Trocear una lista

El troceado de listas funciona de manera totalmente análoga al *troceado de cadenas*. Veamos algunos ejemplos:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping[0:3]
['Agua', 'Huevos', 'Aceite']

>>> shopping[:3]
['Agua', 'Huevos', 'Aceite']

>>> shopping[2:4]
['Aceite', 'Sal']
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> shopping[-1:-4:-1]
['Limón', 'Sal', 'Aceite']

>>> # Equivale a invertir la lista
>>> shopping[::-1]
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

En el troceado de listas, a diferencia de lo que ocurre al obtener elementos, no debemos preocuparnos por acceder a *índices inválidos* (fuera de rango) ya que Python los restringirá a los límites de la lista:

```
>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping[10:]
[]

>>> shopping[-100:2]
['Agua', 'Huevos']

>>> shopping[2:100]
['Aceite', 'Sal', 'Limón']
```

---

**Importante:** Ninguna de las operaciones anteriores modifican la lista original, simplemente devuelven una lista nueva.

---

## Invertir una lista

Python nos ofrece, al menos, tres mecanismos para invertir los elementos de una lista:

**Conservando la lista original:** Mediante *troceado* de listas con *step* negativo:

```
>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping[::-1]
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

**Conservando la lista original:** Mediante la función `reversed()`:

```
>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> list(reversed(shopping))
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

**Modificando la lista original:** Utilizando la función `reverse()` (nótese que es sin «*d*» al final):

```
>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.reverse()

>>> shopping
['Limón', 'Sal', 'Aceite', 'Huevos', 'Agua']
```

### Añadir al final de la lista

Una de las operaciones más utilizadas en listas es añadir elementos al final de las mismas. Para ello Python nos ofrece la función `append()`. Se trata de un método *destructivo* que modifica la lista original:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping.append('Atún')

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Atún']
```

### Creando desde vacío

Una forma muy habitual de trabajar con listas es empezar con una vacía e ir añadiendo elementos poco a poco. Se podría hablar de un **patrón creación**.

Supongamos un ejemplo en el que queremos construir una lista con los números pares del 1 al 20:

```
>>> even_numbers = []

>>> for i in range(20):
...     if i % 2 == 0:
...         even_numbers.append(i)
... 
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> even_numbers
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/2fiS9Ax>

## Añadir en cualquier posición de una lista

Ya hemos visto cómo añadir elementos al final de una lista. Sin embargo, Python ofrece una función `insert()` que vendría a ser una generalización de la anterior, para incorporar elementos en cualquier posición. Simplemente debemos especificar el índice de inserción y el elemento en cuestión. También se trata de una función *destruktiva*<sup>2</sup>:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping.insert(1, 'Jamón')

>>> shopping
['Agua', 'Jamón', 'Huevos', 'Aceite']

>>> shopping.insert(3, 'Queso')

>>> shopping
['Agua', 'Jamón', 'Huevos', 'Queso', 'Aceite']
```

---

**Nota:** El índice que especificamos en la función `insert()` lo podemos interpretar como la posición *delante* (a la izquierda) de la cual vamos a colocar el nuevo valor en la lista.

---

Al igual que ocurriría con el *troceado de listas*, en este tipo de inserciones no obtendremos un error si especificamos índices fuera de los límites de la lista. Estos se ajustarán al principio o al final en función del valor que indiquemos:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping.insert(100, 'Mermelada')

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Mermelada']
```

(continué en la próxima página)

---

<sup>2</sup> Cuando hablamos de que una función/método es «destruktiva/o» significa que modifica la lista (objeto) original, no que la destruye.

(proviene de la página anterior)

```
>>> shopping.insert(-100, 'Arroz')

>>> shopping
['Arroz', 'Agua', 'Huevos', 'Aceite', 'Mermelada']
```

---

**Consejo:** Aunque es posible utilizar `insert()` para añadir **elementos al final de una lista**, siempre se recomienda usar `append()` por su mayor legibilidad:

```
>>> values = [1, 2, 3]
>>> values.append(4)
>>> values
[1, 2, 3, 4]

>>> values = [1, 2, 3]
>>> values.insert(len(values), 4) # don't do it!
>>> values
[1, 2, 3, 4]
```

---

## Repetir elementos

Al igual que con las *cadenas de texto*, el operador `*` nos permite repetir los elementos de una lista:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping * 3
['Agua',
 'Huevos',
 'Aceite',
 'Agua',
 'Huevos',
 'Aceite',
 'Agua',
 'Huevos',
 'Aceite']
```



## Combinar listas

Python nos ofrece dos aproximaciones para combinar listas:

**Conservando la lista original:** Mediante el operador `+` o `+=`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']
>>> fruitshop = ['Naranja', 'Manzana', 'Piña']

>>> shopping + fruitshop
['Agua', 'Huevos', 'Aceite', 'Naranja', 'Manzana', 'Piña']
```

**Modificando la lista original:** Mediante la función `extend()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']
>>> fruitshop = ['Naranja', 'Manzana', 'Piña']

>>> shopping.extend(fruitshop)

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Naranja', 'Manzana', 'Piña']
```

Hay que tener en cuenta que `extend()` funciona adecuadamente si pasamos una **lista como argumento**. En otro caso, quizás los resultados no sean los esperados. Veamos un ejemplo:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping.extend('Limón')

>>> shopping
['Agua', 'Huevos', 'Aceite', 'L', 'i', 'm', 'ó', 'n']
```

El motivo es que `extend()` «recorre» (o itera) sobre cada uno de los elementos del objeto en cuestión. En el caso anterior, al ser una cadena de texto, está formada por caracteres. De ahí el resultado que obtenemos.

Se podría pensar en el uso de `append()` para combinar listas. La realidad es que no funciona exactamente como esperamos; la segunda lista se añadiría como una *sublista* de la principal:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> fruitshop = ['Naranja', 'Manzana', 'Piña']

>>> shopping.append(fruitshop)

>>> shopping
['Agua', 'Huevos', 'Aceite', ['Naranja', 'Manzana', 'Piña']]
```

### Modificar una lista

Del mismo modo que se *accede a un elemento* utilizando su índice, también podemos modificarlo:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite']

>>> shopping[0]
'Agua'

>>> shopping[0] = 'Jugo'

>>> shopping
['Jugo', 'Huevos', 'Aceite']
```

En el caso de acceder a un *índice no válido* de la lista, incluso para modificar, obtendremos un error:

```
>>> shopping[100] = 'Chocolate'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

### Modificar con troceado

No sólo es posible modificar un elemento de cada vez, sino que podemos asignar valores a trozos de una lista:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping[1:4]
['Huevos', 'Aceite', 'Sal']

>>> shopping[1:4] = ['Atún', 'Pasta']

>>> shopping
['Agua', 'Atún', 'Pasta', 'Limón']
```

---

**Nota:** La lista que asignamos no necesariamente debe tener la misma longitud que el trozo que sustituimos.

---

## Borrar elementos

Python nos ofrece, al menos, cuatro formas para borrar elementos en una lista:

**Por su índice:** Mediante la función `del()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> del(shopping[3])

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Limón']
```

**Por su valor:** Mediante la función `remove()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.remove('Sal')

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Limón']
```

**Advertencia:** Si existen valores duplicados, la función `remove()` sólo borrará la primera ocurrencia.

**Por su índice (con extracción):** Las dos funciones anteriores `del()` y `remove()` efectivamente borran el elemento indicado de la lista, pero no «devuelven»<sup>3</sup> nada. Sin embargo, Python nos ofrece la función `pop()` que además de borrar, nos «recupera» el elemento; algo así como una *extracción*. Lo podemos ver como una combinación de *acceso* + *borrado*:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.pop()
'Limón'

>>> shopping
['Agua', 'Huevos', 'Aceite', 'Sal']

>>> shopping.pop(2)
'Aceite'
```

(continué en la próxima página)

<sup>3</sup> Más adelante veremos el comportamiento de las funciones. Devolver o retornar un valor es el resultado de aplicar una función.

(proviene de la página anterior)

```
>>> shopping
['Agua', 'Huevos', 'Sal']
```

---

**Nota:** Si usamos la función `pop()` sin pasarle ningún argumento, por defecto usará el índice `-1`, es decir, el último elemento de la lista. Pero también podemos indicarle el índice del elemento a extraer.

---

**Por su rango:** Mediante troceado de listas:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping[1:4] = []

>>> shopping
['Agua', 'Limón']
```

### Borrado completo de la lista

Python nos ofrece, al menos, dos formas para borrar una lista por completo:

1. Utilizando la función `clear()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.clear() # Borrado in-situ

>>> shopping
[]
```

2. «Reinicializando» la lista a vacío con `[]`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping = [] # Nueva zona de memoria

>>> shopping
[]
```

---

**Nota:** La diferencia entre ambos métodos tiene que ver con cuestiones internas de gestión de memoria y de rendimiento.

---

## Encontrar un elemento

Si queremos descubrir el **índice** que corresponde a un determinado valor dentro la lista podemos usar la función `index()` para ello:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.index('Huevos')
1
```

Tener en cuenta que si el elemento que buscamos no está en la lista, obtendremos un error:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.index('Pollo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'Pollo' is not in list
```

---

**Nota:** Si buscamos un valor que existe más de una vez en una lista, la función `index()` sólo nos devolverá el índice de la primera ocurrencia.

---

## Pertenencia de un elemento

Si queremos comprobar la existencia de un determinado elemento en una lista, podríamos buscar su índice, pero la forma **pitónica** de hacerlo es utilizar el operador `in`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> 'Aceite' in shopping
True

>>> 'Pollo' in shopping
False
```

---

**Nota:** El operador `in` siempre devuelve un valor booleano, es decir, verdadero o falso.

---

## Ejercicio

Determine si una cadena de texto dada es un **isograma**, es decir, no se repite ninguna letra.

Ejemplos válidos de isogramas:

- *lumberjacks*
  - *background*
  - *downstream*
  - *six-year-old*
- 

### Número de ocurrencias

Para contar cuántas veces aparece un determinado valor dentro de una lista podemos usar la función `count()`:

```
>>> sheldon_greeting = ['Penny', 'Penny', 'Penny']

>>> sheldon_greeting.count('Howard')
0

>>> sheldon_greeting.count('Penny')
3
```

### Convertir lista a cadena de texto

Dada una lista, podemos convertirla a una cadena de texto, uniendo todos sus elementos mediante algún **separador**. Para ello hacemos uso de la función `join()` con la siguiente estructura:

```
'=' . join(mylist)
```

Separador                      Lista

Figura 1: Estructura de llamada a la función `join()`

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> ', '.join(shopping)
'Agua,Huevos,Aceite,Sal,Limón'
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> ' '.join(shopping)
'Agua Huevos Aceite Sal Limón'

>>> '|'.join(shopping)
'Agua|Huevos|Aceite|Sal|Limón'
```

Hay que tener en cuenta que `join()` sólo funciona si *todos sus elementos son cadenas de texto*:

```
>>> ', '.join([1, 2, 3, 4, 5])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 0: expected str instance, int found
```

---

**Truco:** Esta función `join()` es realmente la **opuesta** a la de `split()` para *dividir una cadena*.

---

---

## Ejercicio

Consiga la siguiente transformación:

12/31/20 ➡ 31-12-2020

---

## Ordenar una lista

Python proporciona, al menos, dos formas de ordenar los elementos de una lista:

**Conservando lista original:** Mediante la función `sorted()` que devuelve una nueva lista ordenada:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> sorted(shopping)
['Aceite', 'Agua', 'Huevos', 'Limón', 'Sal']
```

**Modificando la lista original:** Mediante la función `sort()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> shopping.sort()
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> shopping
['Aceite', 'Agua', 'Huevos', 'Limón', 'Sal']
```

**Ambos métodos** admiten un *parámetro* «booleano» `reverse` para indicar si queremos que la ordenación se haga en **sentido inverso**:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> sorted(shopping, reverse=True)
['Sal', 'Limón', 'Huevos', 'Agua', 'Aceite']
```

### Longitud de una lista

Podemos conocer el número de elementos que tiene una lista con la función `len()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> len(shopping)
5
```

### Iterar sobre una lista

Al igual que *hemos visto con las cadenas de texto*, también podemos *iterar* sobre los elementos de una lista utilizando la sentencia `for`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> for product in shopping:
...     print(product)
...
Agua
Huevos
Aceite
Sal
Limón
```

---

**Nota:** También es posible usar la sentencia `break` en este tipo de bucles para abortar su ejecución en algún momento que nos interese.

---



## Iterar usando enumeración

Hay veces que no sólo nos interesa «visitar» cada uno de los elementos de una lista, sino que también queremos **saber su índice** dentro de la misma. Para ello Python nos ofrece la función `enumerate()`:

```
>>> shopping = ['Agua', 'Huevos', 'Aceite', 'Sal', 'Limón']

>>> for i, product in enumerate(shopping):
...     print(i, product)
...
0 Agua
1 Huevos
2 Aceite
3 Sal
4 Limón
```

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/TfiuIZ0>

## Iterar sobre múltiples listas

Python ofrece la posibilidad de iterar sobre **múltiples listas en paralelo** utilizando la función `zip()`:

```
>>> shopping = ['Agua', 'Aceite', 'Arroz']
>>> details = ['mineral natural', 'de oliva virgen', 'basmati']

>>> for product, detail in zip(shopping, details):
...     print(product, detail)
...
Agua mineral natural
Aceite de oliva virgen
Arroz basmati
```

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/lfiolG>

---

**Nota:** En el caso de que las listas no tengan la misma longitud, la función `zip()` realiza la combinación hasta que se agota la lista más corta.

---

Dado que `zip()` produce un *iterador*, si queremos obtener una **lista explícita** con la combinación en paralelo de las listas, debemos construir dicha lista de la siguiente manera:

```
>>> shopping = ['Agua', 'Aceite', 'Arroz']
>>> details = ['mineral natural', 'de oliva virgen', 'basmati']

>>> list(zip(shopping, details))
[('Agua', 'mineral natural'),
 ('Aceite', 'de oliva virgen'),
 ('Arroz', 'basmati')]
```

---

### Ejercicio

Dados dos vectores (listas) de la misma dimensión, utilice la función `zip()` para calcular su producto escalar.

### Ejemplo

- Entrada:

```
v1 = [4, 3, 8, 1]
v2 = [9, 2, 7, 3]
```

- Salida: 101

$$v1 \times v2 = [4 \cdot 9 + 3 \cdot 2 + 8 \cdot 7 + 1 \cdot 3] = 101$$

---

## 5.1.4 Cuidado con las copias

### Nivel intermedio

Las listas son estructuras de datos *mutables* y esta característica nos obliga a tener cuidado cuando realizamos copias de listas, ya que la modificación de una de ellas puede afectar a la otra.

Veamos un ejemplo sencillo:

```
>>> original_list = [4, 3, 7, 1]

>>> copy_list = original_list

>>> original_list[0] = 15

>>> original_list
[15, 3, 7, 1]

>>> copy_list
[15, 3, 7, 1]
```

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/pfi5PC5>

---

**Nota:** A través de *Python Tutor* se puede ver claramente el motivo de por qué ocurre esto. Dado que las variables «apuntan» a la misma zona de memoria, al modificar una de ellas, el cambio también se ve reflejado en la otra.

---

Una **posible solución** a este problema es hacer una «copia dura». Para ello Python proporciona la función `copy()`:

```
>>> original_list = [4, 3, 7, 1]

>>> copy_list = original_list.copy()

>>> original_list[0] = 15

>>> original_list
[15, 3, 7, 1]

>>> copy_list
[4, 3, 7, 1]
```

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/Dfi6oLk>

---

**Truco:** En el caso de que estemos trabajando con listas que contienen elementos mutables, debemos hacer uso de la función `deepcopy()` dentro del módulo `copy` de la librería estándar.

---

### 5.1.5 Veracidad múltiple

Si bien podemos usar *sentencias condicionales* para comprobar la veracidad de determinadas expresiones, Python nos ofrece dos funciones «built-in» con las que podemos evaluar si se cumplen **todas** las condiciones `all()` o si se cumple **alguna** condición `any()`. Estas funciones trabajan sobre iterables, y el caso más evidente es una **lista**.

Supongamos un ejemplo en el que queremos comprobar si una determinada palabra cumple las siguientes condiciones:

- Su longitud total es mayor que 4.
- Empieza por «p».
- Contiene, al menos, una «y».

Veamos la **versión clásica**:

```
>>> word = 'python'

>>> if len(word) > 4 and word.startswith('p') and word.count('y') >= 1:
...     print('Cool word!')
... else:
...     print('No thanks')
...
Cool word!
```

Veamos la **versión con veracidad múltiple** usando `all()`, donde se comprueba que se cumplan **todas** las expresiones:

```
>>> word = 'python'

>>> enough_length = len(word) > 4           # True
>>> right_beginning = word.startswith('p')   # True
>>> min_ys = word.count('y') >= 1           # True

>>> is_cool_word = all([enough_length, right_beginning, min_ys])

>>> if is_cool_word:
...     print('Cool word!')
... else:
...     print('No thanks')
...
Cool word!
```

Veamos la **versión con veracidad múltiple** usando `any()`, donde se comprueba que se cumpla **alguna** expresión:

```
>>> word = 'yeah'

>>> enough_length = len(word) > 4           # False
>>> right_beginning = word.startswith('p')   # False
>>> min_ys = word.count('y') >= 1           # True

>>> is_fine_word = any([enough_length, right_beginning, min_ys])

>>> if is_fine_word:
...     print('Fine word!')
... else:
...     print('No thanks')
...
Fine word!
```

**Consejo:** Este enfoque puede ser interesante cuando se manejan muchas condiciones o bien cuando queremos separar las condiciones y agruparlas en una única lista.

### 5.1.6 Listas por comprensión

#### Nivel intermedio

Las **listas por comprensión** establecen una técnica para crear listas de forma más **compacta** basándose en el concepto matemático de **conjuntos definidos por comprensión**.

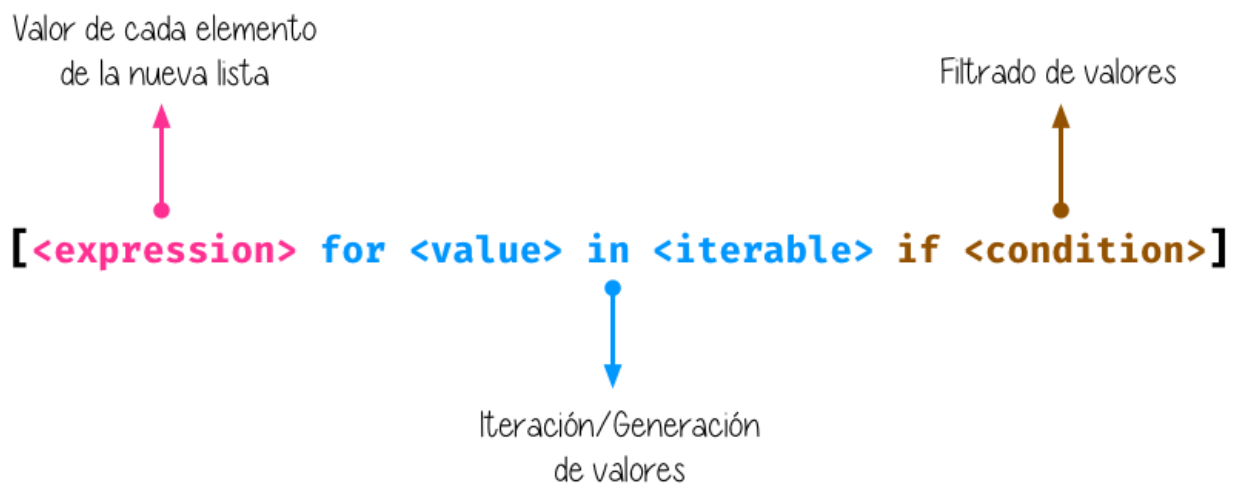


Figura 2: Estructura de una lista por comprensión

En primer lugar veamos un ejemplo en el que convertimos una cadena de texto con valores numéricos en una lista con los mismos valores pero convertidos a enteros. En su **versión clásica** haríamos algo tal que así:

```
>>> values = '32,45,11,87,20,48'

>>> int_values = []

>>> for value in values.split(','):
...     int_value = int(value)
...     int_values.append(int_value)
...

>>> int_values
[32, 45, 11, 87, 20, 48]
```

Ahora veamos el código utilizando una **lista por comprensión**:

```
>>> values = '32,45,11,87,20,48'

>>> int_values = [int(value) for value in values.split(',')]

>>> int_values
[32, 45, 11, 87, 20, 48]
```

### Condiciones en comprensiones

También existe la posibilidad de incluir condiciones en las **listas por comprensión**.

Continuando con el ejemplo anterior, supongamos que sólo queremos crear la lista con aquellos valores que empiecen por el dígito 4:

```
>>> values = '32,45,11,87,20,48'

>>> int_values = [int(v) for v in values.split(',') if v.startswith('4')]

>>> int_values
[45, 48]
```

### Anidamiento en comprensiones

#### Nivel avanzado

En la iteración que usamos dentro de la lista por comprensión es posible usar *bucles anidados*.

Veamos un ejemplo en el que generamos todas las combinaciones de una serie de valores:

```
>>> values = '32,45,11,87,20,48'
>>> svalues = values.split(',')

>>> combinations = [f'{v1}x{v2}' for v1 in svalues for v2 in svalues]

>>> combinations
['32x32',
 '32x45',
 '32x11',
 '32x87',
 '32x20',
 '32x48',
 '45x32',
 '45x45',
 ...]
```

(continué en la próxima página)

(proviene de la página anterior)

```
'48x45',
'48x11',
'48x87',
'48x20',
'48x48']
```

**Consejo:** Las listas por comprensión son una herramienta muy potente y nos ayuda en muchas ocasiones, pero hay que tener cuidado de no generar **expresiones excesivamente complejas**. En estos casos es mejor una *aproximación clásica*.

### Ejercicio

Utilizando listas por comprensión, cree una lista que contenga el resultado de aplicar la función  $f(x) = 3x + 2$  para  $x \in [0, 20)$ .

**Salida esperada:** [2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41, 44, 47, 50, 53, 56, 59]

### 5.1.7 sys.argv

Cuando queramos ejecutar un programa Python desde **línea de comandos**, tendremos la posibilidad de acceder a los argumentos de dicho programa. Para ello se utiliza una lista que la encontramos dentro del módulo `sys` y que se denomina `argv`:

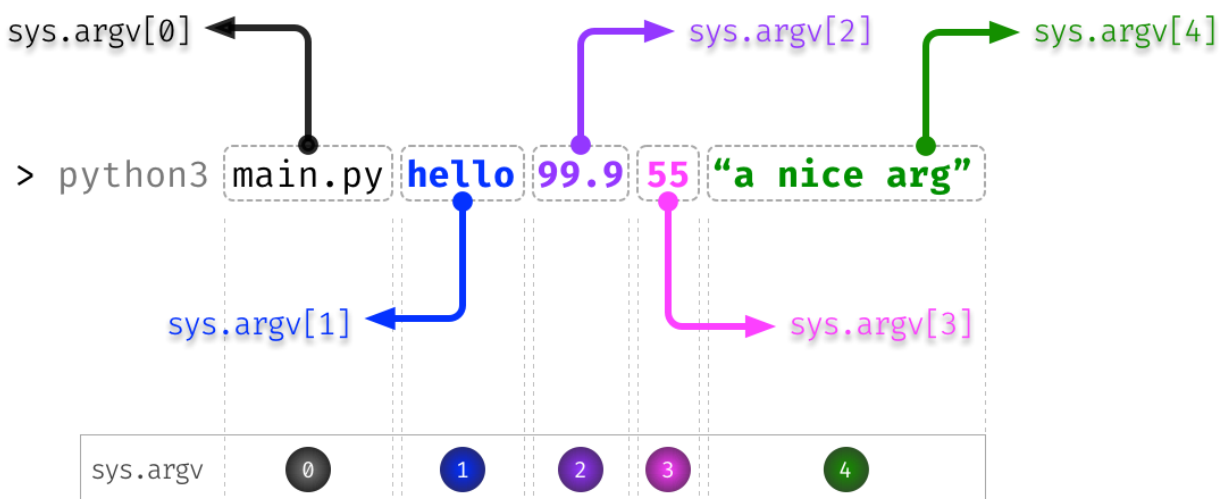


Figura 3: Acceso a parámetros en línea de comandos

Veamos un ejemplo de código en el que simulamos el paso de parámetros recogido en la figura anterior:

get-args.py

```
1 import sys
2
3 filename = sys.argv[0]
4 arg1 = sys.argv[1]
5 arg2 = float(sys.argv[2])
6 arg3 = int(sys.argv[3])
7 arg4 = sys.argv[4]
8
9 print(f'{arg1=}')
10 print(f'{arg2=}')
11 print(f'{arg3=}')
12 print(f'{arg4=}')
```

Si lo ejecutamos obtenemos lo siguiente:

```
$ python3 get-args.py hello 99.9 55 "a nice arg"

arg1='hello'
arg2=99.9
arg3=55
arg4='a nice arg'
```

### 5.1.8 Funciones matemáticas

Python nos ofrece, entre otras<sup>4</sup>, estas tres funciones matemáticas básicas que se pueden aplicar sobre listas.

**Suma de todos los valores:** Mediante la función `sum()`:

```
>>> data = [5, 3, 2, 8, 9, 1]
>>> sum(data)
28
```

**Mínimo de todos los valores:** Mediante la función `min()`:

```
>>> data = [5, 3, 2, 8, 9, 1]
>>> min(data)
1
```

---

<sup>4</sup> Existen multitud de paquetes científicos en Python para trabajar con listas o vectores numéricos. Una de las más famosas es la librería `Numpy`.



**Máximo de todos los valores:** Mediante la función `max()`:

```
>>> data = [5, 3, 2, 8, 9, 1]
>>> max(data)
9
```

---

## Ejercicio

Lea *desde línea de comandos* una serie de números y obtenga la media de dichos valores (*muestre el resultado con 2 decimales*).

La llamada se haría de la siguiente manera:

```
$ python3 avg.py 32 56 21 99 12 17
```

Plantilla de código para el programa:

```
import sys

# En values tendremos una lista con los valores (como strings)
values = sys.argv[1:]

# Su código debajo de aquí
```

## Ejemplo

- Entrada: 32 56 21 99 12 17
  - Salida: 39.50
- 

## 5.1.9 Listas de listas

### Nivel intermedio

Como ya hemos visto en varias ocasiones, las listas son estructuras de datos que pueden contener elementos heterogéneos. Estos elementos pueden ser a su vez listas.

A continuación planteamos un ejemplo del mundo deportivo. Un equipo de fútbol suele tener una disposición en el campo organizada en líneas de jugadores. En aquella alineación con la que España *ganó la copa del mundo* en 2010 había una disposición *4-3-3* con los siguientes jugadores:

Veamos una posible representación de este equipo de fútbol usando una lista compuesta de listas. Primero definimos cada una de las líneas:



Figura 4: Lista de listas (como equipo de fútbol)

```
>>> goalkeeper = 'Casillas'
>>> defenders = ['Capdevila', 'Piqué', 'Puyol', 'Ramos']
>>> midfielders = ['Xabi', 'Busquets', 'X. Alonso']
>>> forwards = ['Iniesta', 'Villa', 'Pedro']
```

Y ahora las juntamos en una única lista:

```
>>> team = [goalkeeper, defenders, midfielders, forwards]

>>> team
['Casillas',
 ['Capdevila', 'Piqué', 'Puyol', 'Ramos'],
 ['Xabi', 'Busquets', 'X. Alonso'],
 ['Iniesta', 'Villa', 'Pedro']]
```

Podemos comprobar el acceso a distintos elementos:

```
>>> team[0] # portero
'Casillas'

>>> team[1][0] # lateral izquierdo
'Capdevila'

>>> team[2] # centrocampistas
['Xabi', 'Busquets', 'X. Alonso']

>>> team[3][1] # delantero centro
'Villa'
```

## Ejercicio

Escriba un programa que permita multiplicar únicamente matrices de 2 filas por 2 columnas. Veamos un ejemplo concreto:

```
A = [[6, 4], [8, 9]]
B = [[3, 2], [1, 7]]
```

El producto  $\mathbb{P} = A \times B$  se calcula siguiendo la [multiplicación de matrices](#) tal y como se indica a continuación:

$$\mathbb{P} = \begin{pmatrix} 6_{[00]} & 4_{[01]} \\ 8_{[10]} & 9_{[11]} \end{pmatrix} \times \begin{pmatrix} 3_{[00]} & 2_{[01]} \\ 1_{[10]} & 7_{[11]} \end{pmatrix} =$$

$$\begin{pmatrix} 6 \cdot 3 + 4 \cdot 1 & 6 \cdot 2 + 4 \cdot 7 \\ 8 \cdot 3 + 9 \cdot 1 & 8 \cdot 2 + 9 \cdot 7 \end{pmatrix} = \begin{pmatrix} 22 & 40 \\ 33 & 79 \end{pmatrix}$$

### EJERCICIOS DE REPASO

1. Escriba un programa en Python que acepte una lista de valores numéricos y obtenga su valor máximo **sin utilizar** la función «built-in» `max()` (**solución**).

Entrada: [6, 3, 9, 2, 10, 31, 15, 7]

Salida: 31

2. Escriba un programa en Python que acepte una lista y elimine sus elementos duplicados (**solución**).

Entrada: ["this", "is", "a", "real", "real", "real", "story"]

Salida: ["this", "is", "a", "real", "story"]

3. Escriba un programa en Python que acepte una lista – que puede contener sublistas (sólo en 1 nivel de anidamiento) – y genere otra lista «aplanada» (**solución**).

Entrada: [0, 10, [20, 30], 40, 50, [60, 70, 80], [90, 100, 110, 120]]

Salida: [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120]

4. Escriba un programa en Python que acepte una lista y genere otra lista eliminando los elementos duplicados consecutivos (**solución**).

Entrada: [0, 0, 1, 2, 3, 4, 4, 5, 6, 6, 6, 7, 8, 9, 4, 4]

Salida: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 4]

5. Escriba un programa en Python que acepte una lista de listas representando una matriz numérica y compute la suma de los elementos de la diagonal principal (**solución**).

Entrada: [[4, 6, 1], [2, 9, 3], [1, 7, 7]]

Salida: 20

### AMPLIAR CONOCIMIENTOS

- [Linked Lists in Python: An Introduction](#)
- [Python Command Line Arguments](#)
- [Sorting Data With Python](#)
- [When to Use a List Comprehension in Python](#)
- [Using the Python `zip\(\)` Function for Parallel Iteration](#)
- [Lists and Tuples in Python](#)
- [How to Use `sorted\(\)` and `sort\(\)` in Python](#)
- [Using List Comprehensions Effectively](#)

## 5.2 Tuplas



El concepto de **tupla** es muy similar al de *lista*. Aunque hay algunas diferencias menores, lo fundamental es que, mientras una *lista* es mutable y se puede modificar, una *tupla* no admite cambios y por lo tanto, es **inmutable**.<sup>1</sup>

### 5.2.1 Creando tuplas

Podemos pensar en crear tuplas tal y como *lo hacíamos con listas*, pero usando **paréntesis** en lugar de *corchetes*:

```
>>> empty_tuple = ()  
  
>>> tenerife_geoloc = (28.46824, -16.25462)  
  
>>> three_wise_men = ('Melchor', 'Gaspar', 'Baltasar')
```

<sup>1</sup> Foto original de portada por [engin akyurt](#) en Unsplash.

### Tuplas de un elemento

Hay que prestar especial atención cuando vamos a crear una **tupla de un único elemento**. La intención primera sería hacerlo de la siguiente manera:

```
>>> one_item_tuple = ('Papá Noel')

>>> one_item_tuple
'Papá Noel'

>>> type(one_item_tuple)
str
```

Realmente, hemos creado una variable de tipo **str** (cadena de texto). Para crear una tupla de un elemento debemos añadir una **coma** al final:

```
>>> one_item_tuple = ('Papá Noel',)

>>> one_item_tuple
('Papá Noel',)

>>> type(one_item_tuple)
tuple
```

### Tuplas sin paréntesis

Según el caso, hay veces que nos podemos encontrar con tuplas que no llevan paréntesis. Quizás no está tan extendido, pero a efectos prácticos tiene el mismo resultado. Veamos algunos ejemplos de ello:

```
>>> one_item_tuple = 'Papá Noel',

>>> three_wise_men = 'Melchor', 'Gaspar', 'Baltasar'

>>> tenerife_geoloc = 28.46824, -16.25462
```

### 5.2.2 Modificar una tupla

Como ya hemos comentado previamente, las tuplas con estructuras de datos **inmutables**. Una vez que las creamos con un valor, no podemos modificarlas. Veamos qué ocurre si lo intentamos:

```
>>> three_wise_men = 'Melchor', 'Gaspar', 'Baltasar'

>>> three_wise_men[0] = 'Tom Hanks'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

### 5.2.3 Conversión

Para convertir otros tipos de datos en una tupla podemos usar la función `tuple()`:

```
>>> shopping = ['Agua', 'Aceite', 'Arroz']

>>> tuple(shopping)
('Agua', 'Aceite', 'Arroz')
```

Esta conversión es válida para aquellos tipos de datos que sean *iterables*: cadenas de caracteres, listas, diccionarios, conjuntos, etc. Un ejemplo que no funciona es intentar convertir un número en una tupla:

```
>>> tuple(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
```

El uso de la función `tuple()` sin argumentos equivale a crear una tupla vacía:

```
>>> tuple()
()
```

---

**Truco:** Para crear una tupla vacía, se suele recomendar el uso de `()` frente a `tuple()`, no sólo por ser más *pitónico* sino por tener (en promedio) un mejor rendimiento en tiempos de ejecución.

---

## 5.2.4 Operaciones con tuplas

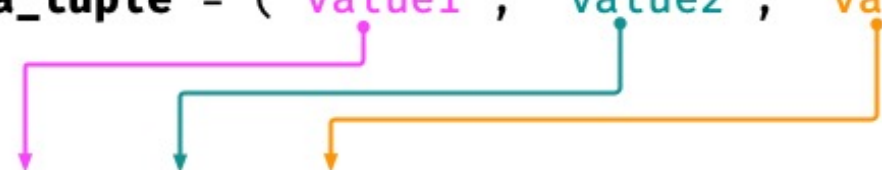
Con las tuplas podemos realizar *todas las operaciones que vimos con listas* salvo las que conlleven una modificación «in-situ» de la misma:

- `reverse()`
- `append()`
- `extend()`
- `remove()`
- `clear()`
- `sort()`

## 5.2.5 Desempaquetado de tuplas

El **desempaquetado** es una característica de las tuplas que nos permite *asignar una tupla a variables independientes*:

```
>>> a_tuple = ('value1', 'value2', 'value3')
```



```
>>> var1, var2, var3 = a_tuple
```

También es una tupla

Figura 5: Desempaquetado de tuplas

Veamos un ejemplo con código:

```
>>> three_wise_men = ('Melchor', 'Gaspar', 'Baltasar')

>>> king1, king2, king3 = three_wise_men

>>> king1
'Melchor'
>>> king2
'Gaspar'
```

(continué en la próxima página)



(proviene de la página anterior)

```
>>> king3
'Baltasar'
```

Python proporciona la función «built-in» `divmod()` que devuelve el cociente y el resto de una división usando una única llamada. Lo interesante (para el caso que nos ocupa) es que se suele utilizar el desempaquetado de tuplas para obtener los valores:

```
>>> quotient, remainder = divmod(7, 3)

>>> quotient
2
>>> remainder
1
```

## Intercambio de valores

A través del desempaquetado de tuplas podemos llevar a cabo *el intercambio de los valores de dos variables* de manera directa:

```
>>> value1 = 40
>>> value2 = 20

>>> value1, value2 = value2, value1

>>> value1
20
>>> value2
40
```

---

**Nota:** A priori puede parecer que esto es algo «natural», pero en la gran mayoría de lenguajes de programación no es posible hacer este intercambio de forma «directa» ya que necesitamos recurrir a una tercera variable «auxiliar» como almacén temporal en el paso intermedio de traspaso de valores.

---

### Desempaquetado extendido

No tenemos que ceñirnos a realizar desempaquetado uno a uno. También podemos extenderlo e indicar ciertos «grupos» de elementos mediante el operador `*`.

Veamos un ejemplo:

```
>>> ranking = ('G', 'A', 'R', 'Y', 'W')
>>> head, *body, tail = ranking
>>> head
'G'
>>> body
['A', 'R', 'Y']
>>> tail
'W'
```

### Desempaquetado genérico

El desempaquetado de tuplas es extensible a cualquier tipo de datos que sea **iterable**. Veamos algunos ejemplos de ello.

Sobre cadenas de texto:

```
>>> oxygen = 'O2'
>>> first, last = oxygen
>>> first, last
('O', '2')
>>> text = 'Hello, World!'
>>> head, *body, tail = text
>>> head, body, tail
('H', ['e', 'l', 'l', 'o', ',', ' ', 'W', 'o', 'r', 'l', 'd'], '!')
```

Sobre listas:

```
>>> writer1, writer2, writer3 = ['Virginia Woolf', 'Jane Austen', 'Mary Shelley']
>>> writer1, writer2, writer3
('Virginia Woolf', 'Jane Austen', 'Mary Shelley')
>>> text = 'Hello, World!'
>>> word1, word2 = text.split()
>>> word1, word2
('Hello', 'World!')
```

### 5.2.6 ¿Tuplas por comprensión?

Los tipos de datos mutables (*listas, diccionarios y conjuntos*) sí permiten comprensiones pero no así los tipos de datos inmutables como *cadenas de texto* y *tuplas*.

Si intentamos crear una **tupla por comprensión** utilizando paréntesis alrededor de la expresión, vemos que no obtenemos ningún error al ejecutarlo:

```
>>> myrange = (number for number in range(1, 6))
```

Sin embargo no hemos conseguido una tupla por comprensión sino un generador:

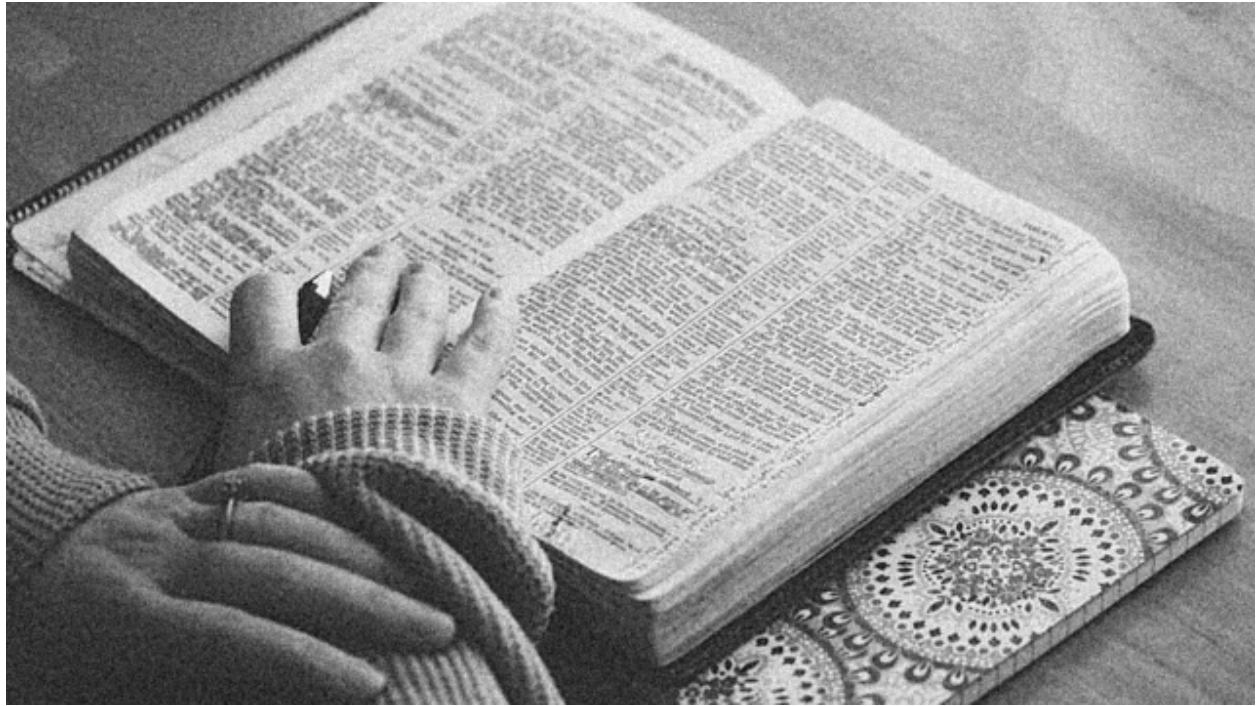
```
>>> myrange
<generator object <genexpr> at 0x10b3732e0>
```

### 5.2.7 Tuplas vs Listas

Aunque puedan parecer estructuras de datos muy similares, sabemos que las tuplas carecen de ciertas operaciones, especialmente las que tienen que ver con la modificación de sus valores, ya que no son inmutables. Si las listas son más flexibles y potentes, ¿por qué íbamos a necesitar tuplas? Veamos 4 potenciales ventajas del uso de tuplas frente a las listas:

1. Las tuplas ocupan **menos espacio** en memoria.
2. En las tuplas existe **protección** frente a cambios indeseados.
3. Las tuplas se pueden usar como **claves de diccionarios** (son «*hashables*»).
4. Las `namedtuples` son una alternativa sencilla a los objetos.

## 5.3 Diccionarios



Podemos trasladar el concepto de *diccionario* de la vida real al de *diccionario* en Python. Al fin y al cabo un diccionario es un objeto que contiene palabras, y cada palabra tiene asociado un significado. Haciendo el paralelismo, diríamos que en Python un diccionario es también un objeto indexado por **claves** (las palabras) que tienen asociados unos **valores** (los significados).<sup>1</sup>

Los diccionarios en Python tienen las siguientes *características*:

- Mantienen el **orden** en el que se insertan las claves.<sup>2</sup>
- Son **mutables**, con lo que admiten añadir, borrar y modificar sus elementos.
- Las **claves** deben ser **únicas**. A menudo se utilizan las *cadenas de texto* como claves, pero en realidad podría ser cualquier tipo de datos inmutable: enteros, flotantes, tuplas (entre otros).
- Tienen un **acceso muy rápido** a sus elementos, debido a la forma en la que están implementados internamente.<sup>3</sup>

---

**Nota:** En otros lenguajes de programación, a los diccionarios se les conoce como *arrays*

---

<sup>1</sup> Foto original de portada por [Aaron Burden](#) en Unsplash.

<sup>2</sup> Aunque históricamente Python no establecía que las claves de los diccionarios tuvieran que mantener su orden de inserción, a partir de Python 3.7 este comportamiento cambió y se garantizó el orden de inserción de las claves como [parte oficial de la especificación del lenguaje](#).

<sup>3</sup> Véase este [análisis de complejidad y rendimiento](#) de distintas estructuras de datos en CPython.

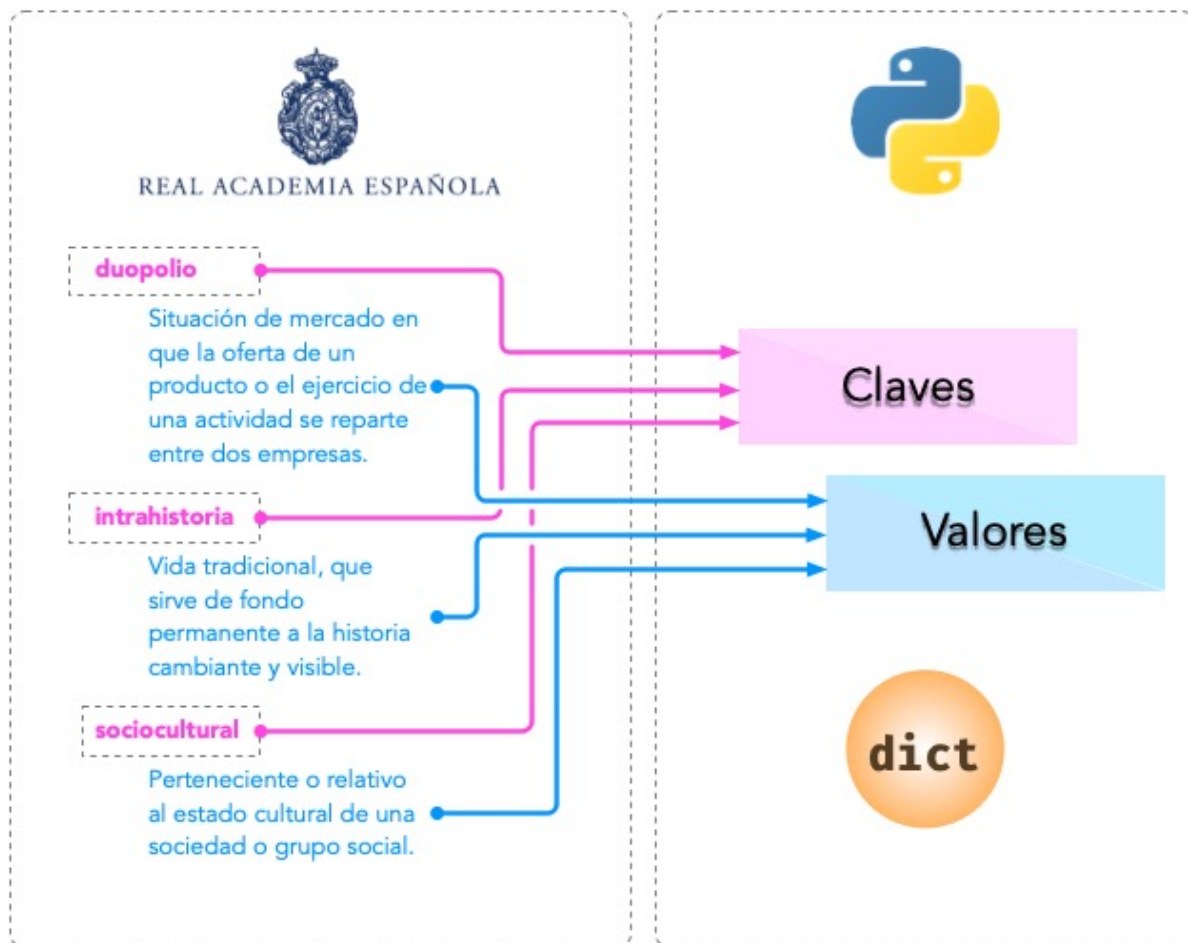


Figura 6: Analogía de un diccionario en Python

*asociativos, «hashes» o «hashmaps».*

---

### 5.3.1 Creando diccionarios

Para crear un diccionario usamos llaves {} rodeando asignaciones **clave: valor** que están separadas por comas. Veamos algunos ejemplos de diccionarios:

```
>>> empty_dict = {}

>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }

>>> population_can = {
...     2015: 2_135_209,
...     2016: 2_154_924,
...     2017: 2_177_048,
...     2018: 2_206_901,
...     2019: 2_220_270
... }
```

En el código anterior podemos observar la creación de un diccionario vacío, otro donde sus claves y sus valores son cadenas de texto y otro donde las claves y los valores son valores enteros.

Ejecución **paso a paso** a través de *Python Tutor*:

<https://cutt.ly/Sfav2Yw>

---

### Ejercicio

Cree un diccionario con los nombres de 5 personas de su familia y sus edades.

---

### 5.3.2 Conversión

Para convertir otros tipos de datos en un diccionario podemos usar la función `dict()`:

```
>>> # Diccionario a partir de una lista de cadenas de texto
>>> dict(['a1', 'b2'])
{'a': '1', 'b': '2'}
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> # Diccionario a partir de una tupla de cadenas de texto
>>> dict(('a1', 'b2'))
{'a': '1', 'b': '2'}

>>> # Diccionario a partir de una lista de listas
>>> dict(['a', 1], ['b', 2])
{'a': 1, 'b': 2}
```

---

**Nota:** Si nos fijamos bien, cualquier iterable que tenga una estructura interna de 2 elementos es susceptible de convertirse en un diccionario a través de la función `dict()`.

---

## Diccionario vacío

Existe una manera particular de usar `dict()` y es no pasarle ningún argumento. En este caso estaremos queriendo convertir el «vacío» en un diccionario, con lo que obtendremos un *diccionario vacío*:

```
>>> dict()
{}
```

---

**Truco:** Para crear un diccionario vacío, se suele recomendar el uso de `{}` frente a `dict()`, no sólo por ser más *pitónico* sino por tener (en promedio) un mejor rendimiento en tiempos de ejecución.

---

## Creación con `dict()`

También es posible utilizar la función `dict()` para crear diccionarios y no tener que utilizar llaves y comillas:

Supongamos que queremos transformar la siguiente tabla en un diccionario:

Atributo	Valor
name	Guido
surname	Van Rossum
job	Python creator

Utilizando la construcción mediante `dict` podemos pasar clave y valor como **argumentos** de la función:

```
>>> person = dict(
...     name='Guido',
...     surname='Van Rossum',
...     job='Python creator'
... )

>>> person
{'name': 'Guido', 'surname': 'Van Rossum', 'job': 'Python creator'}
```

El inconveniente que tiene esta aproximación es que las **claves deben ser identificadores válidos** en Python. Por ejemplo, no se permiten espacios:

```
>>> person = dict(
...     name='Guido van Rossum',
...     date of birth='31/01/1956'
File "<stdin>", line 3
    date of birth='31/01/1956'
        ^
SyntaxError: invalid syntax
```

### Nivel intermedio

Es posible crear un diccionario especificando sus claves y un único valor de «relleno»:

```
>>> dict.fromkeys('aeiou', 0)
{'a': 0, 'e': 0, 'i': 0, 'o': 0, 'u': 0}
```

---

**Nota:** Es válido pasar cualquier «iterable» como referencia a las claves.

---

## 5.3.3 Operaciones con diccionarios

### Obtener un elemento

Para obtener un elemento de un diccionario basta con escribir la **clave** entre corchetes. Veamos un ejemplo:

```
>>> rae = {
...     'bifronte': 'De dos frentes o dos caras',
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa'
... }
```

(continué en la próxima página)



(proviene de la página anterior)

```
>>> rae['anarcoide']  
'Que tiende al desorden'
```

Si intentamos acceder a una clave que no existe, obtendremos un error:

```
>>> rae['acceso']  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
KeyError: 'acceso'
```

## Usando get()

Existe una función muy útil para «superar» los posibles errores de acceso por claves inexistentes. Se trata de `get()` y su comportamiento es el siguiente:

1. Si la clave que buscamos existe, nos devuelve su valor.
2. Si la clave que buscamos no existe, nos devuelve `None`<sup>4</sup> salvo que le indiquemos otro valor por defecto, pero en ninguno de los dos casos obtendremos un error.

```
1 >>> rae  
2 {'bifronte': 'De dos frentes o dos caras',  
3  'anarcoide': 'Que tiende al desorden',  
4  'montuvio': 'Campesino de la costa'}  
5  
6 >>> rae.get('bifronte')  
7 'De dos frentes o dos caras'  
8  
9 >>> rae.get('programación')  
10  
11 >>> rae.get('programación', 'No disponible')  
12 'No disponible'
```

**Línea 6:** Equivalente a `rae['bifronte']`.

**Línea 9:** La clave buscada no existe y obtenemos `None`.<sup>5</sup>

**Línea 11:** La clave buscada no existe y nos devuelve el valor que hemos aportado por defecto.

<sup>4</sup> `None` es la palabra reservada en Python para la «nada». Más información en [esta web](#).

<sup>5</sup> Realmente no estamos viendo nada en la consola de Python porque la representación en cadena de texto es vacía.

### Añadir o modificar un elemento

Para añadir un elemento a un diccionario sólo es necesario hacer referencia a la *clave* y asignarle un *valor*:

- Si la clave **ya existía** en el diccionario, **se reemplaza** el valor existente por el nuevo.
- Si la clave **es nueva**, **se añade** al diccionario con su valor. *No vamos a obtener un error a diferencia de las listas.*

Partimos del siguiente diccionario para ejemplificar estas acciones:

```
>>> rae = {  
...     'bifronte': 'De dos frentes o dos caras',  
...     'anarcoide': 'Que tiende al desorden',  
...     'montuvio': 'Campesino de la costa'  
... }
```

Vamos a **añadir** la palabra *enjuiciar* a nuestro diccionario de la Real Academia de La Lengua:

```
>>> rae['enjuiciar'] = 'Someter una cuestión a examen, discusión y juicio'  
  
>>> rae  
{'bifronte': 'De dos frentes o dos caras',  
 'anarcoide': 'Que tiende al desorden',  
 'montuvio': 'Campesino de la costa',  
 'enjuiciar': 'Someter una cuestión a examen, discusión y juicio'}
```

Supongamos ahora que queremos **modificar** el significado de la palabra *enjuiciar* por otra acepción:

```
>>> rae['enjuiciar'] = 'Instruir, juzgar o sentenciar una causa'  
  
>>> rae  
{'bifronte': 'De dos frentes o dos caras',  
 'anarcoide': 'Que tiende al desorden',  
 'montuvio': 'Campesino de la costa',  
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa'}
```

---

### Ejercicio

Construya un diccionario partiendo de una cadena de texto con el siguiente formato:

<city>:<population>;<city>:<population>;<city>:<population>;....

- Claves: **ciudades**.
- Valores: **habitantes** (*como enteros*).

## Ejemplo

- Entrada: Tokyo:38\_140\_000;Delhi:26\_454\_000;Shanghai:24\_484\_000;Mumbai:21\_357\_000;São Paulo:21\_297\_000
  - Salida: {'Tokyo': 38140000, 'Delhi': 26454000, 'Shanghai': 24484000, 'Mumbai': 21357000, 'São Paulo': 21297000}
- 

## Creando desde vacío

Una forma muy habitual de trabajar con diccionarios es utilizar el **patrón creación** partiendo de uno vacío e ir añadiendo elementos poco a poco.

Supongamos un ejemplo en el que queremos construir un diccionario donde las claves son las letras vocales y los valores son sus posiciones:

```
>>> VOWELS = 'aeiou'

>>> enum_vowels = {}

>>> for i, vowel in enumerate(VOWELS):
...     enum_vowels[vowel] = i + 1
...

>>> enum_vowels
{'a': 1, 'e': 2, 'i': 3, 'o': 4, 'u': 5}
```

---

**Nota:** Hemos utilizando la función `enumerate()` que ya vimos para las listas en el apartado: *Iterar usando enumeración*.

---

## Pertenencia de una clave

La forma **pitónica** de comprobar la existencia de una clave dentro de un diccionario, es utilizar el operador `in`:

```
>>> 'bifronte' in rae
True

>>> 'almohada' in rae
False

>>> 'montuvio' not in rae
False
```

---

**Nota:** El operador `in` siempre devuelve un valor booleano, es decir, verdadero o falso.

---

### Ejercicio

Usando un diccionario, cuente el número de veces que se repite cada letra en una cadena de texto dada.

### Ejemplo

- Entrada: 'boom'
  - Salida: {'b': 1, 'o': 2, 'm': 1}
- 

### Obtener todos los elementos

Python ofrece mecanismos para obtener todos los elementos de un diccionario. Partimos del siguiente diccionario:

```
>>> rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa'}
```

**Obtener todas las claves de un diccionario:** Mediante la función `keys()`:

```
>>> rae.keys()
dict_keys(['bifronte', 'anarcoide', 'montuvio', 'enjuiciar'])
```

**Obtener todos los valores de un diccionario:** Mediante la función `values()`:

```
>>> rae.values()
dict_values([
    'De dos frentes o dos caras',
    'Que tiende al desorden',
    'Campesino de la costa',
    'Instruir, juzgar o sentenciar una causa'
])
```

**Obtener todos los pares «clave-valor» de un diccionario:** Mediante la función `items()`:

```
>>> rae.items()
dict_items([
    ('bifronte', 'De dos frentes o dos caras'),
    ('anarcoide', 'Que tiende al desorden'),
    ('montuvio', 'Campesino de la costa'),
    ('enjuiciar', 'Instruir, juzgar o sentenciar una causa')
])
```

**Nota:** Para este último caso cabe destacar que los «items» se devuelven como una lista de *tuplas*, donde cada tupla tiene dos elementos: el primero representa la clave y el segundo representa el valor.

## Longitud de un diccionario

Podemos conocer el número de elementos («clave-valor») que tiene un diccionario con la función `len()`:

```
>>> rae
{'bifronte': 'De dos frentes o dos caras',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa'}

>>> len(rae)
4
```

## Iterar sobre un diccionario

En base a *los elementos que podemos obtener*, Python nos proporciona tres maneras de iterar sobre un diccionario.

### Iterar sobre claves:

```
>>> for word in rae.keys():
...     print(word)
...
bifronte
anarcoide
montuvio
enjuiciar
```

### Iterar sobre valores:

```
>>> for meaning in rae.values():  
...     print(meaning)  
...  
De dos frentes o dos caras  
Que tiende al desorden  
Campesino de la costa  
Instruir, juzgar o sentenciar una causa
```

Iterar sobre «clave-valor»:

```
>>> for word, meaning in rae.items():  
...     print(f'{word}: {meaning}')  
...  
bifronte: De dos frentes o dos caras  
anarcoide: Que tiende al desorden  
montuvio: Campesino de la costa  
enjuiciar: Instruir, juzgar o sentenciar una causa
```

---

**Nota:** En este último caso, recuerde el uso de los *«f-strings»* para formatear cadenas de texto.

---

---

### Ejercicio

Dado el diccionario de ciudades y poblaciones ya visto, y suponiendo que estas ciudades son las únicas que existen en el planeta, calcule el porcentaje de población relativo de cada una de ellas con respecto al total.

### Ejemplo

- Entrada: Tokyo:38\_140\_000;Delhi:26\_454\_000;Shanghai:24\_484\_000;  
Mumbai:21\_357\_000;São Paulo:21\_297\_000
  - Salida: {'Tokyo': 28.952722193544467, 'Delhi': 20.081680988673973,  
'Shanghai': 18.58622050830474, 'Mumbai': 16.212461664591746, 'São Paulo':  
16.16691464488507}
-

## Combinar diccionarios

Dados dos (o más) diccionarios, es posible «mezclarlos» para obtener una combinación de los mismos. Esta combinación se basa en dos premisas:

1. Si la clave no existe, se añade con su valor.
2. Si la clave ya existe, se añade con el valor del «último» diccionario en la mezcla.<sup>6</sup>

Python ofrece dos mecanismos para realizar esta combinación. Vamos a partir de los siguientes diccionarios para ejemplificar su uso:

```
>>> rae1 = {
...     'bifronte': 'De dos frentes o dos caras',
...     'enjuiciar': 'Someter una cuestión a examen, discusión y juicio'
... }

>>> rae2 = {
...     'anarcoide': 'Que tiende al desorden',
...     'montuvio': 'Campesino de la costa',
...     'enjuiciar': 'Instruir, juzgar o sentenciar una causa'
... }
```

**Sin modificar los diccionarios originales:** Mediante el operador `**`:

```
>>> {**rae1, **rae2}
{'bifronte': 'De dos frentes o dos caras',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}
```

A partir de **Python 3.9** podemos utilizar el operador `|` para combinar dos diccionarios:

```
>>> rae1 | rae2
{'bifronte': 'De dos frentes o dos caras',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa',
 'anarcoide': 'Que tiende al desorden',
 'montuvio': 'Campesino de la costa'}
```

**Modificando los diccionarios originales:** Mediante la función `update()`:

```
>>> rae1.update(rae2)

>>> rae1
{'bifronte': 'De dos frentes o dos caras',
 'enjuiciar': 'Instruir, juzgar o sentenciar una causa',
```

(continué en la próxima página)

<sup>6</sup> En este caso «último» hace referencia al diccionario que se encuentra más a la derecha en la expresión.

(proviene de la página anterior)

```
'anarcoide': 'Que tiende al desorden',  
'montuvio': 'Campesino de la costa'}
```

---

**Nota:** Tener en cuenta que el orden en el que especificamos los diccionarios a la hora de su combinación (mezcla) es relevante en el resultado final. En este caso *el orden de los factores sí altera el producto*.

---

### Borrar elementos

Python nos ofrece, al menos, tres formas para borrar elementos en un diccionario:

**Por su clave:** Mediante la sentencia `del`:

```
>>> rae = {  
...     'bifronte': 'De dos frentes o dos caras',  
...     'anarcoide': 'Que tiende al desorden',  
...     'montuvio': 'Campesino de la costa'  
... }  
  
>>> del(rae['bifronte'])  
  
>>> rae  
{'anarcoide': 'Que tiende al desorden', 'montuvio': 'Campesino de la costa'}
```

**Por su clave (con extracción):** Mediante la función `pop()` podemos extraer un elemento del diccionario por su clave. Vendría a ser una combinación de `get()` + `del`:

```
>>> rae = {  
...     'bifronte': 'De dos frentes o dos caras',  
...     'anarcoide': 'Que tiende al desorden',  
...     'montuvio': 'Campesino de la costa'  
... }  
  
>>> rae.pop('anarcoide')  
'Que tiende al desorden'  
  
>>> rae  
{'bifronte': 'De dos frentes o dos caras', 'montuvio': 'Campesino de la costa'}  
  
>>> rae.pop('bucle')  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
KeyError: 'bucle'
```



**Advertencia:** Si la clave que pretendemos extraer con `pop()` no existe, obtendremos un error.

### Borrado completo del diccionario:

1. Utilizando la función `clear()`:

```
>>> rae = {  
...     'bifronte': 'De dos frentes o dos caras',  
...     'anarcoide': 'Que tiende al desorden',  
...     'montuvio': 'Campesino de la costa'  
... }  
  
>>> rae.clear()  
  
>>> rae  
{}
```

2. «Reinicializando» el diccionario a vacío con `{}`:

```
>>> rae = {  
...     'bifronte': 'De dos frentes o dos caras',  
...     'anarcoide': 'Que tiende al desorden',  
...     'montuvio': 'Campesino de la costa'  
... }  
  
>>> rae = {}  
  
>>> rae  
{}
```

---

**Nota:** La diferencia entre ambos métodos tiene que ver con cuestiones internas de gestión de memoria y de rendimiento.

---

## 5.3.4 Cuidado con las copias

### Nivel intermedio

Al igual que ocurriría con *las listas*, si hacemos un cambio en un diccionario, se verá reflejado en todas las variables que hagan referencia al mismo. Esto se deriva de su propiedad de ser *mutable*. Veamos un ejemplo concreto:

```
>>> original_rae = {  
...     'bifronte': 'De dos frentes o dos caras',  
...     'anarcoide': 'Que tiende al desorden',  
...     'montuvio': 'Campesino de la costa'  
... }  
  
>>> copy_rae = original_rae  
  
>>> original_rae['bifronte'] = 'bla bla bla'  
  
>>> original_rae  
{'bifronte': 'bla bla bla',  
 'anarcoide': 'Que tiende al desorden',  
 'montuvio': 'Campesino de la costa'}  
  
>>> copy_rae  
{'bifronte': 'bla bla bla',  
 'anarcoide': 'Que tiende al desorden',  
 'montuvio': 'Campesino de la costa'}
```

Una **posible solución** a este problema es hacer una «copia dura». Para ello Python proporciona la función `copy()`:

```
>>> original_rae = {  
...     'bifronte': 'De dos frentes o dos caras',  
...     'anarcoide': 'Que tiende al desorden',  
...     'montuvio': 'Campesino de la costa'  
... }  
  
>>> copy_rae = original_rae.copy()  
  
>>> original_rae['bifronte'] = 'bla bla bla'  
  
>>> original_rae  
{'bifronte': 'bla bla bla',  
 'anarcoide': 'Que tiende al desorden',  
 'montuvio': 'Campesino de la costa'}  
  
>>> copy_rae  
{'bifronte': 'De dos frentes o dos caras',  
 'anarcoide': 'Que tiende al desorden',  
 'montuvio': 'Campesino de la costa'}
```

---

**Truco:** En el caso de que estemos trabajando con diccionarios que contienen elementos mutables, debemos hacer uso de la función `deepcopy()` dentro del módulo `copy` de la librería

estándar.

---

### 5.3.5 Diccionarios por comprensión

#### Nivel intermedio

De forma análoga a cómo se escriben las *listas por comprensión*, podemos aplicar este método a los diccionarios usando llaves { }.

Veamos un ejemplo en el que creamos un **diccionario por comprensión** donde las claves son palabras y los valores son sus longitudes:

```
>>> words = ('sun', 'space', 'rocket', 'earth')

>>> words_length = {word: len(word) for word in words}

>>> words_length
{'sun': 3, 'space': 5, 'rocket': 6, 'earth': 5}
```

También podemos aplicar **condiciones** a estas comprensiones. Continuando con el ejemplo anterior, podemos incorporar la restricción de sólo incluir palabras que no empiecen por vocal:

```
>>> words = ('sun', 'space', 'rocket', 'earth')

>>> words_length = {w: len(w) for w in words if w[0] not in 'aeiou'}

>>> words_length
{'sun': 3, 'space': 5, 'rocket': 6}
```

---

**Nota:** Se puede consultar el [PEP-274](#) para ver más ejemplos sobre diccionarios por comprensión.

---

### 5.3.6 Objetos «hashables»

#### Nivel avanzado

La única restricción que deben cumplir las **claves** de un diccionario es ser «**hashables**»<sup>7</sup>. Un objeto es «hashable» si se le puede asignar un valor «hash» que no cambia en ejecución durante toda su vida.

---

<sup>7</sup> Se recomienda [esta ponencia](#) de Víctor Terrón sobre objetos «hashables».

Para encontrar el «hash» de un objeto, Python usa la función `hash()`, que devuelve un número entero y es utilizado para indexar la *tabla «hash»* que se mantiene internamente:

```
>>> hash(999)
999

>>> hash(3.14)
322818021289917443

>>> hash('hello')
-8103770210014465245

>>> hash(('a', 'b', 'c'))
-2157188727417140402
```

Para que un objeto sea «hashable», debe ser **immutable**:

```
>>> hash(['a', 'b', 'c'])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

---

**Nota:** De lo anterior se deduce que las claves de los diccionarios, al tener que ser «hasheables», sólo pueden ser objetos inmutables.

---

La función «built-in» `hash()` realmente hace una llamada al método mágico `__hash__()` del objeto en cuestión:

```
>>> hash('spiderman')
-8105710090476541603

>>> 'spiderman'.__hash__()
-8105710090476541603
```

---

## EJERCICIOS DE REPASO

1. Escriba un programa en Python que acepte una lista de palabras y las agrupe por su letra inicial usando un diccionario (**solución**).

Entrada: [ "mesa", "móvil", "barco", "coche", "avión", "bandeja", "casa",  
"monitor", "carretera", "arco"]

Salida: { "m": [ "mesa", "móvil", "monitor"], "b": [ "barco", "bandeja"], "c":  
[ "coche", "casa", "carretera"], "a": [ "avión", "arco"] }

2. Escriba un programa en Python que acepte un diccionario y determine si todos los valores son iguales o no (**solución**).

Entrada: {"Juan": 5, "Antonio": 5, "Inma": 5, "Ana": 5, "Esteban": 5}

Salida: Same values

3. Escriba un programa en Python que acepte una lista de listas con varios elementos y obtenga un diccionario donde las claves serán los primeros elementos de las sublistas y los valores serán los restantes – como listas – (**solución**).

Entrada: [["Episode IV - A New Hope", "May 25", 1977], ["Episode V - The Empire Strikes Back", "May 21", 1980], ["Episode VI - Return of the Jedi", "May 25", 1983]]

Salida: {"Episode IV - A New Hope": ["May 25", 1977], "Episode V - The Empire Strikes Back": ["May 21", 1980], "Episode VI - Return of the Jedi": ["May 25", 1983]}

4. Escriba un programa en Python que acepte un diccionario cuyos valores son listas y borre el contenido de dichas listas (**solución**).

Entrada: {"C1": [10, 20, 30], "C2": [20, 30, 40], "C3": [12, 34]}

Salida: {"C1": [], "C2": [], "C3": []}

5. Escriba un programa en Python que acepte un diccionario y elimine los espacios de sus claves respetando los valores correspondientes (**solución**).

Entrada: {"S 001": ["Math", "Science"], "S 002": ["Math", "English"]}

Salida: {"S001": ["Math", "Science"], "S002": ["Math", "English"]}

## AMPLIAR CONOCIMIENTOS

- [Using the Python defaultdict Type for Handling Missing Keys](#)
- [Python Dictionary Iteration: Advanced Tips & Tricks](#)
- [Python KeyError Exceptions and How to Handle Them](#)
- [Dictionaries in Python](#)
- [How to Iterate Through a Dictionary in Python](#)
- [Shallow vs Deep Copying of Python Objects](#)

## 5.4 Conjuntos



Un **conjunto** en Python representa una serie de **valores únicos y sin orden establecido**, con la única restricción de que sus elementos deben ser «*hashables*». Mantiene muchas similitudes con el [concepto matemático de conjunto](#)<sup>1</sup>

### 5.4.1 Creando conjuntos

Para crear un conjunto basta con separar sus valores por *comas* y rodearlos de llaves {}:

```
>>> lottery = {21, 10, 46, 29, 31, 94}

>>> lottery
{10, 21, 29, 31, 46, 94}
```

La excepción la tenemos a la hora de crear un **conjunto vacío**, ya que, siguiendo la lógica de apartados anteriores, deberíamos hacerlo a través de llaves:

```
>>> wrong_empty_set = {}

>>> type(wrong_empty_set)
dict
```

---

<sup>1</sup> Foto original de portada por [Duy Pham](#) en Unsplash.

**Advertencia:** Si hacemos esto, lo que obtenemos es un *diccionario vacío*.

La única opción que tenemos es utilizar la función `set()`:

```
>>> empty_set = set()

>>> empty_set
set()

>>> type(empty_set)
set
```

### 5.4.2 Conversión

Para convertir otros tipos de datos en un conjunto podemos usar la función `set()` sobre cualquier iterable:

```
>>> set('aplatanada')
{'a', 'd', 'l', 'n', 'p', 't'}

>>> set([1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5])
{1, 2, 3, 4, 5}

>>> set(('ADENINA', 'TIMINA', 'TIMINA', 'GUANINA', 'ADENINA', 'CITOSINA'))
{'ADENINA', 'CITOSINA', 'GUANINA', 'TIMINA'}

>>> set({'manzana': 'rojo', 'plátano': 'amarillo', 'kiwi': 'verde'})
{'kiwi', 'manzana', 'plátano'}
```

**Importante:** Como se ha visto en los ejemplos anteriores, `set()` se suele utilizar en muchas ocasiones como una forma de **extraer los valores únicos** de otros tipos de datos. En el caso de los diccionarios se extraen las claves, que, por definición, son únicas.

**Nota:** El hecho de que en los ejemplos anteriores los elementos de los conjuntos estén ordenados es únicamente un «detalle de implementación» en el que no se puede confiar.

### 5.4.3 Operaciones con conjuntos

#### Obtener un elemento

En un conjunto no existe un orden establecido para sus elementos, por lo tanto **no podemos acceder a un elemento en concreto**.

De este hecho se deriva igualmente que **no podemos modificar un elemento existente**, ya que ni siquiera tenemos acceso al mismo. Python sí nos permite añadir o borrar elementos de un conjunto.

#### Añadir un elemento

Para añadir un elemento a un conjunto debemos utilizar la función `add()`. Como ya hemos indicado, al no importar el orden dentro del conjunto, la inserción no establece a priori la posición donde se realizará.

A modo de ejemplo, vamos a partir de un conjunto que representa a los cuatro integrantes originales de *The Beatles*. Luego añadiremos a un nuevo componente:

```
>>> # John Lennon, Paul McCartney, George Harrison y Ringo Starr
>>> beatles = set(['Lennon', 'McCartney', 'Harrison', 'Starr'])

>>> beatles.add('Best') # Pete Best

>>> beatles
{'Best', 'Harrison', 'Lennon', 'McCartney', 'Starr'}
```

Ejecución **paso a paso** a través de *Python Tutor*:

<https://tinyurl.com/9folv2v>

---

#### Ejercicio

Dada una tupla de duplas (2 valores), cree dos conjuntos:

- Uno de ellos con los primeros valores de cada dupla.
- El otro con los segundos valores de cada dupla.

#### Ejemplo

- Entrada: ((4, 3), (8, 2), (7, 5), (8, 2), (9, 1))
- Salida:

```
{8, 9, 4, 7}
{1, 2, 3, 5}
```



## Borrar elementos

Para borrar un elemento de un conjunto podemos utilizar la función `remove()`. Siguiendo con el ejemplo anterior, vamos a borrar al último «beatle» añadido:

```
>>> beatles
{'Best', 'Harrison', 'Lennon', 'McCartney', 'Starr'}

>>> beatles.remove('Best')

>>> beatles
{'Harrison', 'Lennon', 'McCartney', 'Starr'}
```

## Longitud de un conjunto

Podemos conocer el número de elementos que tiene un conjunto con la función `len()`:

```
>>> beatles
{'Harrison', 'Lennon', 'McCartney', 'Starr'}

>>> len(beatles)
4
```

## Iterar sobre un conjunto

Tal y como hemos visto para otros tipos de datos *iterables*, la forma de recorrer los elementos de un conjunto es utilizar la sentencia `for`:

```
>>> for beatle in beatles:
...     print(beatle)
...
Harrison
McCartney
Starr
Lennon
```

---

**Consejo:** Como en el ejemplo anterior, es muy común utilizar una *variable en singular* para recorrer un iterable (en plural). No es una regla fija ni sirve para todos los casos, pero sí suele ser una *buena práctica*.

---

### Pertenencia de elemento

Al igual que con otros tipos de datos, Python nos ofrece el operador `in` para determinar si un elemento pertenece a un conjunto:

```
>>> beatles
{'Harrison', 'Lennon', 'McCartney', 'Starr'}

>>> 'Lennon' in beatles
True

>>> 'Fari' in beatles
False
```

### 5.4.4 Teoría de conjuntos

Vamos a partir de dos conjuntos  $A = \{1, 2\}$  y  $B = \{2, 3\}$  para ejemplificar las distintas operaciones que se pueden hacer entre ellos basadas en los [Diagramas de Venn](#) y la [Teoría de Conjuntos](#):

```
>>> A = {1, 2}
>>> B = {2, 3}
```

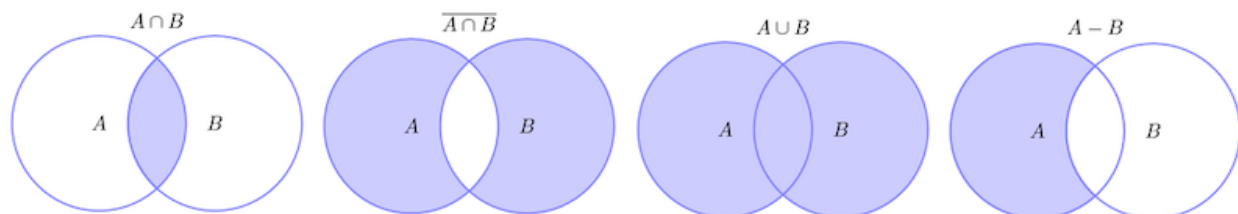


Figura 7: Diagramas de Venn

### Intersección

$A \cap B$  – Elementos que están a la vez en  $A$  y en  $B$ :

```
>>> A & B
{2}

>>> A.intersection(B)
{2}
```

## Unión

$A \cup B$  – Elementos que están tanto en  $A$  como en  $B$ :

```
>>> A | B
{1, 2, 3}

>>> A.union(B)
{1, 2, 3}
```

## Diferencia

$A - B$  – Elementos que están en  $A$  y no están en  $B$ :

```
>>> A - B
{1}

>>> A.difference(B)
{1}
```

## Diferencia simétrica

$\overline{A \cap B}$  – Elementos que están en  $A$  o en  $B$  pero no en ambos conjuntos:

```
>>> A ^ B
{1, 3}

>>> A.symmetric_difference(B)
{1, 3}
```

### 5.4.5 Conjuntos por comprensión

Los conjuntos, al igual que las *listas* y los *diccionarios*, también se pueden crear por comprensión.

Veamos un ejemplo en el que construimos un conjunto por comprensión con los aquellos números enteros múltiplos de 3 en el rango  $[0, 20)$ :

```
>>> m3 = {number for number in range(0, 20) if number % 3 == 0}

>>> m3
{0, 3, 6, 9, 12, 15, 18}
```

### Ejercicio

Dadas dos cadenas de texto, obtenga una nueva cadena de texto con las **letras consonantes** que se **repiten en ambas frases**. Ignore los espacios en blanco y muestre la cadena de salida con sus *letras ordenadas*.

Resuelva el ejercicio mediante dos aproximaciones: Una de ellas usando conjuntos por comprensión y otra sin usar comprensiones.

### Ejemplo

- Entrada: Flat is better than nested y Readability counts
  - Salida: bdlnst
- 

## 5.4.6 Conjuntos inmutables

Python ofrece la posibilidad de crear **conjuntos inmutables** haciendo uso de la función `frozenset()` que recibe cualquier iterable como argumento.

Supongamos que recibimos una serie de calificaciones de exámenes y queremos crear un conjunto inmutable con los posibles niveles (categorías) de calificaciones:

```
>>> marks = [1, 3, 2, 3, 1, 4, 2, 4, 5, 2, 5, 5, 3, 1, 4]

>>> marks_levels = frozenset(marks)

>>> marks_levels
frozenset({1, 2, 3, 4, 5})
```

Veamos qué ocurre si intentamos modificar este conjunto:

```
>>> marks_levels.add(50)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

---

**Nota:** Los `frozenset` son a los `sets` lo que las tuplas a las listas: una forma de «congelar» los valores para que no se puedan modificar.

---

## AMPLIAR CONOCIMIENTOS

- Sets in Python

## 5.5 Ficheros



Aunque los ficheros encajarían más en un apartado de «*entrada/salida*» ya que representan un **medio de almacenamiento persistente**, también podrían ser vistos como *estructuras de datos*, puesto que nos permiten guardar la información y asignarles un cierto formato.<sup>1</sup>

Un **fichero** es un *conjunto de bytes* almacenados en algún *dispositivo*. El *sistema de ficheros* es la estructura lógica que alberga los ficheros y está jerarquizado a través de *directorios* (o carpetas). Cada fichero se identifica unívocamente a través de una *ruta* que nos permite acceder a él.

---

<sup>1</sup> Foto original de portada por [Maksym Kaharlytskyi](#) en Unsplash.

### 5.5.1 Lectura de un fichero

Python ofrece la función `open()` para «abrir» un fichero. Esta apertura se puede realizar en 3 modos distintos:

- **Lectura** del contenido de un fichero existente.
- **Escritura** del contenido en un fichero nuevo.
- **Añadido** al contenido de un fichero existente.

Veamos un ejemplo para leer el contenido de un fichero en el que se encuentran las temperaturas máximas y mínimas de cada día de la última semana. El fichero está en la subcarpeta (*ruta relativa*) `files/temps.dat` y tiene el siguiente contenido:

```
29 23
31 23
34 26
33 23
29 22
28 22
28 22
```

Lo primero será abrir el fichero:

```
>>> f = open('files/temps.dat')
```

La función `open()` recibe como primer argumento la **ruta al fichero** que queremos manejar (como un «string») y devuelve el manejador del fichero, que en este caso lo estamos asignando a una variable llamada `f` pero le podríamos haber puesto cualquier otro nombre.

---

**Nota:** Es importante dominar los conceptos de **ruta relativa** y **ruta absoluta** para el trabajo con ficheros. Véase [este artículo de DeNovatoANovato](#).

---

Hay que tener en cuenta que la ruta al fichero que abrimos (*en modo lectura*) **debe existir**, ya que de lo contrario obtendremos un error:

```
>>> f = open('foo.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'foo.txt'
```

Una vez abierto el fichero ya podemos proceder a leer su contenido. Para ello Python nos ofrece la posibilidad de leer todo el fichero de una vez o bien leerlo línea a línea.

## Lectura completa de un fichero

Siguiendo con nuestro ejemplo de temperaturas, veamos cómo leer todo el contenido del fichero de una sola vez. Para esta operación, Python nos provee, al menos, de dos funciones:

**read()** Devuelve todo el contenido del fichero como una cadena de texto (**str**):

```
>>> f = open('files/temps.dat')

>>> f.read()
'29 23\n31 23\n34 26\n33 23\n29 22\n28 22\n28 22\n'
```

**readlines()** Devuelve todo el contenido del fichero como una lista (**list**) donde cada elemento es una línea:

```
>>> f = open('files/temps.dat')

>>> f.readlines()
['29 23\n', '31 23\n', '34 26\n', '33 23\n', '29 22\n', '28 22\n', '28 22\n']
```

**Importante:** Nótese que, en ambos casos, los saltos de línea `\n` siguen apareciendo en los datos leídos, por lo que habría que «limpiar» estos caracteres. Para ello se recomienda utilizar *las funciones ya vistas de cadenas de texto*.

## Lectura línea a línea

Hay situaciones en las que interesa leer el contenido del fichero línea a línea. Imaginemos un fichero de tamaño considerable (varios GB). Si intentamos leer completamente este fichero de sola una vez podríamos ocupar demasiada RAM y reducir el rendimiento de nuestra máquina.

Es por ello que Python nos ofrece varias aproximaciones a la lectura de ficheros línea a línea. La más usada es **iterar** sobre el propio *manejador* del fichero:

```
>>> f = open('files/temps.dat')

>>> for line in f:    # that easy!
...     print(line)
...
29 23

31 23

34 26
```

(continué en la próxima página)

(proviene de la página anterior)

```
33 23
29 22
28 22
28 22
```

---

**Truco:** Igual que pasaba anteriormente, la lectura línea por línea también incluye el **salto de línea** `\n` lo que provoca un «doble espacio» entre cada una de las salidas. Bastaría con aplicar `line.split()` para eliminarlo.

---

### 5.5.2 Escritura en un fichero

Para escribir texto en un fichero hay que abrir dicho fichero en **modo escritura**. Para ello utilizamos un *argumento adicional* en la función `open()` que indica esta operación:

```
>>> f = open('files/canary-iata.dat', 'w')
```

---

**Nota:** Si bien el fichero en sí mismo se crea al abrirlo en modo escritura, la **ruta** hasta ese fichero no. Eso quiere decir que debemos asegurarnos que **las carpetas hasta llegar a dicho fichero existen**. En otro caso obtenemos un error de tipo `FileNotFoundError`.

---

Ahora ya podemos hacer uso de la función `write()` para enviar contenido al fichero abierto.

Supongamos que queremos volcar el contenido de una lista en dicho fichero. En este caso partimos de los *códigos IATA* de aeropuertos de las Islas Canarias<sup>2</sup>.

```
1 >>> canary_iata = ("GCFV", "GCHI", "GCLA", "GCLP", "GCGM", "GCRR", "GCTS", "GCXO")
2
3 >>> for code in canary_iata:
4 ...     f.write(code + '\n')
5 ...
6
7 >>> f.close()
```

Nótese:

---

<sup>2</sup> Fuente: [Smart Drone](#)



**Línea 4** Escritura de cada código en el fichero. La función `write()` no incluye el salto de línea por defecto, así que lo añadimos de *manera explícita*.

**Línea 7** Cierre del fichero con la función `close()`. Especialmente en el caso de la escritura de ficheros, se recomienda encarecidamente cerrar los ficheros para evitar pérdida de datos.

**Advertencia:** Siempre que se abre un fichero en **modo escritura** utilizando el argumento `'w'`, el fichero se inicializa, borrando cualquier contenido que pudiera tener.

### 5.5.3 Añadido a un fichero

La única diferencia entre añadir información a un fichero y *escribir información en un fichero* es el modo de apertura del fichero. En este caso utilizamos `'a'` por «append»:

```
>>> f = open('more-data.txt', 'a')
```

En este caso el fichero `more-data.txt` se abrirá en *modo añadir* con lo que las llamadas a la función `write()` hará que aparezcan nueva información al final del contenido ya existente en dicho fichero.

### 5.5.4 Usando contextos

Python ofrece *gestores de contexto* como una solución para establecer reglas de entrada y salida a un determinado bloque de código.

En el caso que nos ocupa, usaremos la sentencia `with` y el contexto creado se ocupará de cerrar adecuadamente el fichero que hemos abierto, liberando así sus recursos:

```
1 >>> with open('files/temps.dat') as f:
2 ...     for line in f:
3 ...         max_temp, min_temp = line.strip().split()
4 ...         print(max_temp, min_temp)
5 ...
6 29 23
7 31 23
8 34 26
9 33 23
10 29 22
11 28 22
12 28 22
```

**Línea 1** Apertura del fichero en *modo lectura* utilizando el gestor de contexto definido por la palabra reservada `with`.

**Línea 2** Lectura del fichero línea a línea utilizando la iteración sobre el *manejador del fichero*.

**Línea 3** Limpieza de saltos de línea con `strip()` encadenando la función `split()` para separar las dos temperaturas por el caracter *espacio*. Ver *limpiar una cadena* y *dividir una cadena*.

**Línea 4** Imprimir por pantalla la temperatura mínima y la máxima.

---

**Nota:** Es una buena práctica usar `with` cuando se manejan ficheros. La ventaja es que el fichero se cierra adecuadamente en cualquier circunstancia, incluso si se produce cualquier **tipo de error**.

---

Hay que prestar atención a la hora de escribir valores numéricos en un fichero, ya que el método `write()` por defecto espera ver un «string» como argumento:

```
>>> lottery = [43, 21, 99, 18, 37, 99]

>>> with open('files/lottery.dat', 'w') as f:
...     for number in lottery:
...         f.write(number + '\n')
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

---

**Importante:** Para evitar este tipo de **errores**, se debe convertir a `str` aquellos valores que queramos usar con la función `write()` para escribir información en un fichero de texto.

---

---

### Ejercicio

Dado el fichero `temperatures.txt` con 12 filas (meses) y 31 columnas (temperaturas de cada día), se pide:

1. Leer el fichero de datos.
2. Calcular la temperatura media de cada mes.
3. Escribir un fichero de salida `avgtemps.txt` con 12 filas (*meses*) y la temperatura media de cada mes.

*Guarda el fichero en la misma carpeta en la que vas a escribir tu código. Así evitarás problemas de rutas relativas/absolutas.*

---

## AMPLIAR CONOCIMIENTOS

- Reading and Writing Files in Python
- Python Context Managers and the «with» Statement

