Programación en Python

Clase 2.1: Colecciones de datos

25/03/2026

Motivación

¿Cómo podemos organizar el siguiente conjunto de datos?

```
nombre_1 = "Juan"
nombre_2 = "Carla"
nombre_3 = "Evelina"
nombre_4 = "Ana"

edad_1 = 29
edad_2 = 34
edad_3 = 33
edad_4 = 38
```

Motivación

¿Cómo podemos organizar el siguiente conjunto de datos?

```
nombre_1 = "Juan"
nombre_2 = "Carla"
nombre_3 = "Evelina"
nombre_4 = "Ana"

edad_1 = 29
edad_2 = 34
edad_3 = 33
edad_4 = 38
```

- ¿Qué pasa si la cantidad de datos crece?

Motivación

¿Cómo podemos organizar el siguiente conjunto de datos?

```
nombre_1 = "Juan"
nombre_2 = "Carla"
nombre_3 = "Evelina"
nombre_4 = "Ana"

edad_1 = 29
edad_2 = 34
edad_3 = 33
edad_4 = 38
```

- ¿Qué pasa si la cantidad de datos crece?
- Los tipos de datos elementales son limitados.

Motivación

¿Cómo podemos organizar el siguiente conjunto de datos?

```
nombre_1 = "Juan"
nombre_2 = "Carla"
nombre_3 = "Evelina"
nombre_4 = "Ana"

edad_1 = 29
edad_2 = 34
edad_3 = 33
edad_4 = 38
```

- ¿Qué pasa si la cantidad de datos crece?
- Los tipos de datos elementales son limitados.
- Necesitamos estructuras de datos más complejas.

Hoy

- Listas
- Tuplas
- Diccionarios

Listas

Definición

Esto es una lista

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

Definición

Esto es una lista

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

Esto también

```
>>> ["a", "e", True, 256]
['a', 'e', True, 256]
```

Definición

Esto es una lista

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

Esto también

```
>>> ["a", "e", True, 256]
['a', 'e', True, 256]
```

¿Pero qué es?

- Secuencia ordenada de objetos
- Mutable

Definición

Esto es una lista

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

Esto también

```
>>> ["a", "e", True, 256]
['a', 'e', True, 256]
```

¿Pero qué es?

- Secuencia ordenada de objetos
- Mutable

Objeto que contiene otros objetos en un orden determinado, cuyo contenido puede modificarse.

Creación

Las listas se definen escribiendo elementos entre corchetes `[]`, separados por comas:

```
>>> lista = ["Bayes", "Laplace", "Fisher"]
```

Creación

Las listas se definen escribiendo elementos entre corchetes `[]`, separados por comas:

```
>>> lista = ["Bayes", "Laplace", "Fisher"]
```

Su representación es idéntica a la definición:

```
>>> lista
['Bayes', 'Laplace', 'Fisher']
```

Creación

Las listas se definen escribiendo elementos entre corchetes `[]`, separados por comas:

```
>>> lista = ["Bayes", "Laplace", "Fisher"]
```

Su representación es idéntica a la definición:

```
>>> lista
['Bayes', 'Laplace', 'Fisher']
```

Y su tipo es `list`:

```
>>> type(lista)
<class 'list'>
```

¿Qué quiere decir que es ordenada?

Supongamos las siguientes listas:

```
>>> l1 = [1, 2, 3]
>>> l2 = [2, 1, 3]
```

¿Qué quiere decir que es ordenada?

Supongamos las siguientes listas:

```
>>> l1 = [1, 2, 3]
>>> l2 = [2, 1, 3]
```

¿Son iguales?

```
>>> l1 == l2
False
```

¿Qué quiere decir que es ordenada?

Y estas otras listas, ¿son iguales?

```
>>> l1 = [1, 2, 3]
>>> l2 = [1, 2, 3]
```

¿Qué quiere decir que es ordenada?

Y estas otras listas, ¿son iguales?

```
>>> l1 = [1, 2, 3]
>>> l2 = [1, 2, 3]
```

```
>>> l1 == l2
True
```

¿Qué quiere decir que es ordenada?

Y estas otras listas, ¿son iguales?

```
>>> l1 = [1, 2, 3]
>>> l2 = [1, 2, 3]
```

```
>>> l1 == l2
True
```

¿Y son idénticas?

¿Qué quiere decir que es ordenada?

Y estas otras listas, ¿son iguales?

```
>>> l1 = [1, 2, 3]
>>> l2 = [1, 2, 3]
```

```
>>> l1 == l2
True
```

¿Y son idénticas?

```
>>> l1 is l2
False
```

Igualdad en valor y en identidad

Mientras `==` compara valor, `is` compara identidad.

Igualdad en valor y en identidad

Mientras `==` compara valor, `is` compara identidad.

```
>>> l1 == l2
True
```

Igualdad en valor y en identidad

Mientras `==` compara valor, `is` compara identidad.

```
>>> l1 == l2
True
>>> l1 is l2
False
```

Igualdad en valor y en identidad

Mientras `==` compara valor, `is` compara identidad.

```
>>> l1 == l2
True
>>> l1 is l2
False
>>> id(l1)
127087380616448
>>> id(l2)
127087378339712
```

Igualdad en valor y en identidad

Mientras `==` compara valor, `is` compara identidad.

```
>>> l1 == l2
True
>>> l1 is l2
False
>>> id(l1)
127087380616448
>>> id(l2)
127087378339712
```

Dos objetos pueden ser **iguales** pero no **el mismo**.

Detalles de implementación

Con escalares inmutables y cadenas de texto, Python puede reutilizar objetos en memoria.

```
>>> 10 is 10
True
```

Detalles de implementación

Con escalares inmutables y cadenas de texto, Python puede reutilizar objetos en memoria.

```
>>> 10 is 10
True
```

```
>>> "hola" is "hola"
True
```

Detalles de implementación

Con escalares inmutables y cadenas de texto, Python puede reutilizar objetos en memoria.

```
>>> 10 is 10
True
```

```
>>> "hola" is "hola"
True
```

```
>>> True is True
True
```

Detalles de implementación

Con escalares inmutables y cadenas de texto, Python puede reutilizar objetos en memoria.

```
>>> 10 is 10
True
```

```
>>> "hola" is "hola"
True
```

```
>>> True is True
True
```

Dado que Python almacena estos valores una única vez en memoria, la comparación en identidad devuelve `True`.

Detalles de implementación

Con escalares inmutables y cadenas de texto, Python puede reutilizar objetos en memoria.

```
>>> 10 is 10
True
```

```
>>> "hola" is "hola"
True
```

```
>>> True is True
True
```

Dado que Python almacena estos valores una única vez en memoria, la comparación en identidad devuelve `True`.

¿Cuál es la razón para almacenarlos una única vez?

Acceder a elementos

Se accede a un elemento pasando su índice dentro corchetes [ ]:

Acceder a elementos

Se accede a un elemento pasando su índice dentro corchetes `[]`:

```
>>> autores = ["Agresti", "Dobson", "Gelman"]
```

Acceder a elementos

Se accede a un elemento pasando su índice dentro corchetes `[]`:

```
>>> autores = ["Agresti", "Dobson", "Gelman"]
>>> autores[1]
```

Acceder a elementos

Se accede a un elemento pasando su índice dentro corchetes `[]`:

```
>>> autores = ["Agresti", "Dobson", "Gelman"]
>>> autores[1]
'Dobson'
```

Acceder a elementos

Se accede a un elemento pasando su índice dentro corchetes `[]`:

```
>>> autores = ["Agresti", "Dobson", "Gelman"]
>>> autores[1]
'Dobson'
```

¿Por qué obtuvimos `Dobson`, si es el segundo elemento?

Zero-based indexing

Python, como muchos lenguajes, inicia sus índices en 0, no en 1.

Zero-based indexing

Python, como muchos lenguajes, inicia sus índices en 0, no en 1.

```
>>> autores[0]
'Agresti'
>>> autores[1]
'Dobson'
>>> autores[2]
'Gelman'
```

Zero-based indexing

Python, como muchos lenguajes, inicia sus índices en 0, no en 1.

```
>>> autores[0]
'Agresti'
>>> autores[1]
'Dobson'
>>> autores[2]
'Gelman'
```

En síntesis

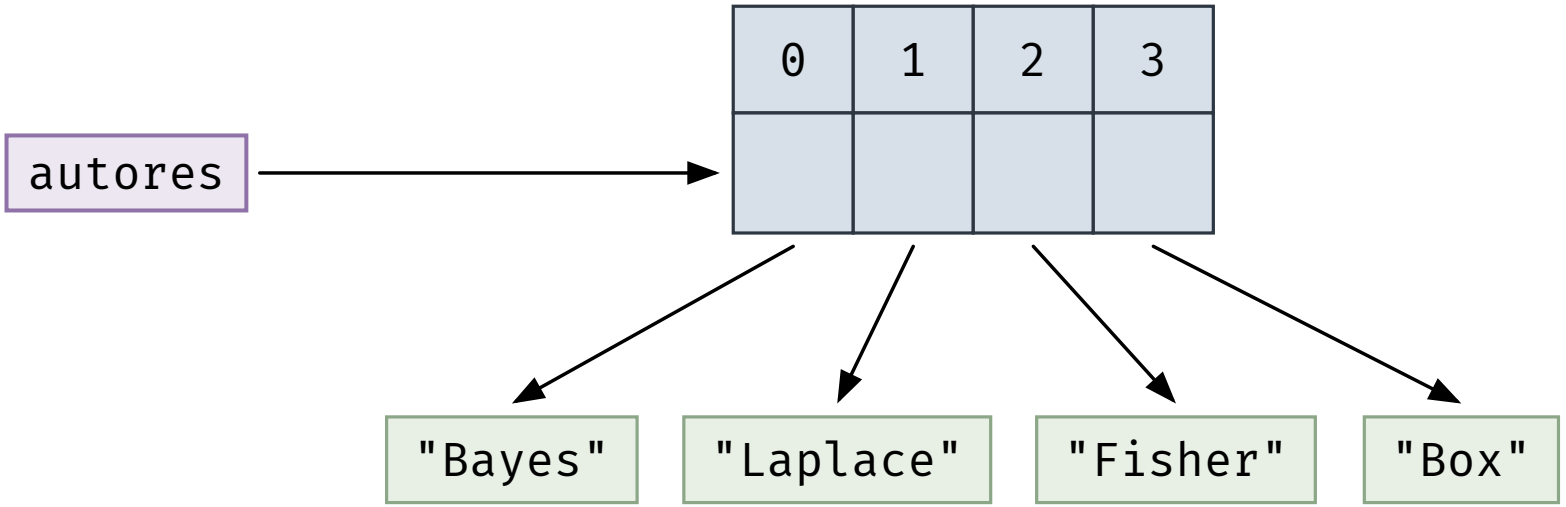
- El primer elemento está en la posición **0**.
- El último elemento está en la posición **n - 1**.

Selección de elementos

```
>>> autores = ["Bayes", "Laplace", "Fisher", "Box"]
```

Selección de elementos

```
>>> autores = ["Bayes", "Laplace", "Fisher", "Box"]
```

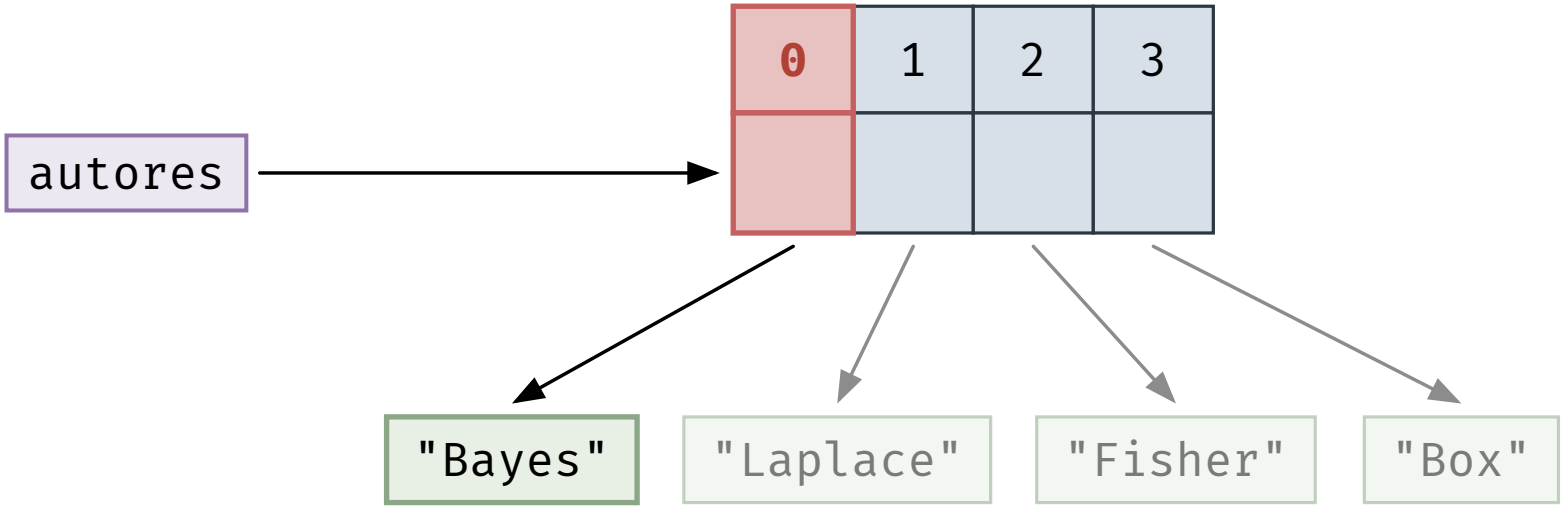


Selección de elementos

```
>>> autores[0]
```

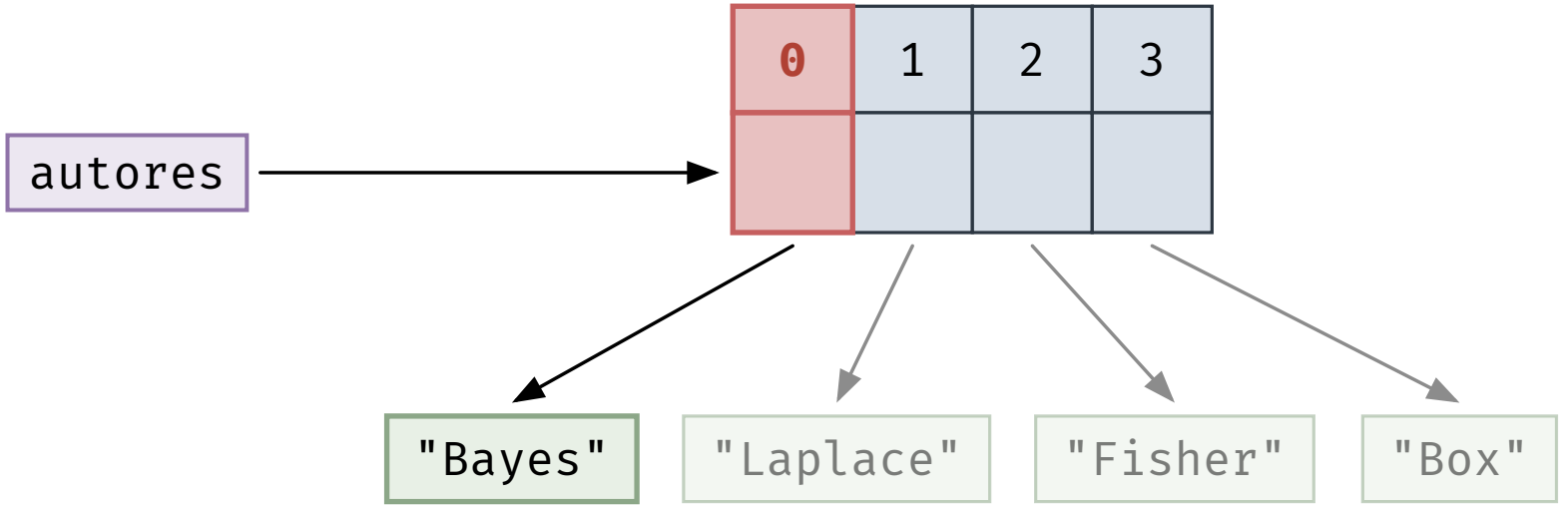
Selección de elementos

```
>>> autores[0]
```



Selección de elementos

```
>>> autores[0]
```



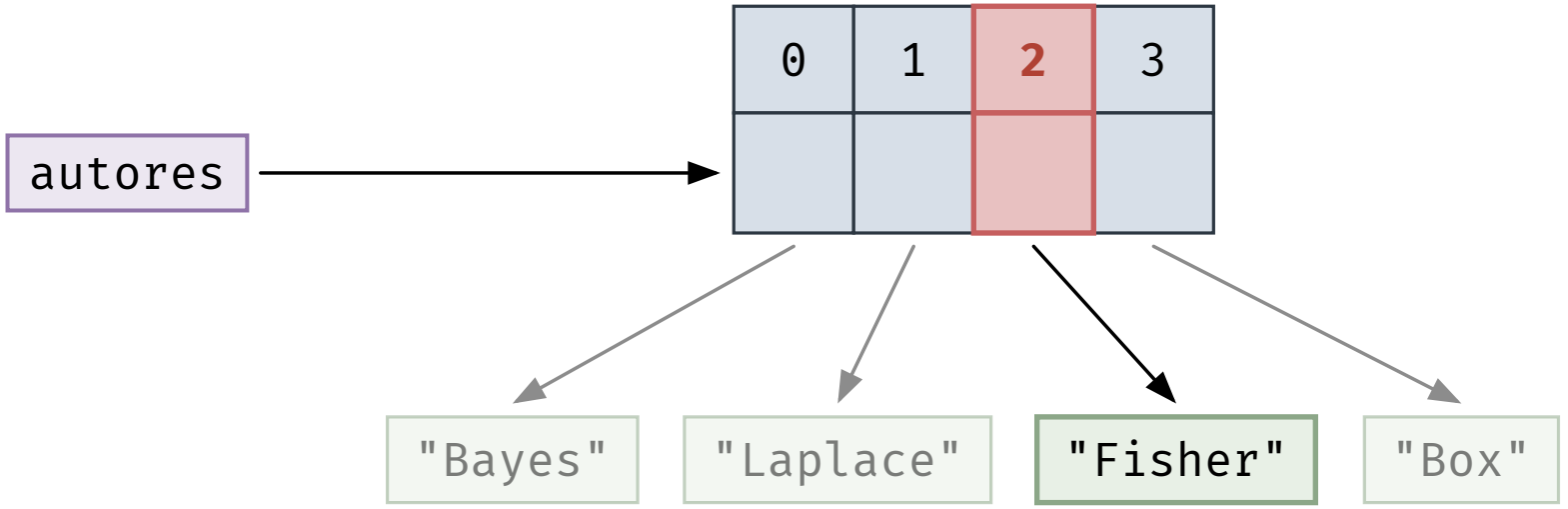

```
'Bayes'
```

Selección de elementos

```
>>> autores[2]
```

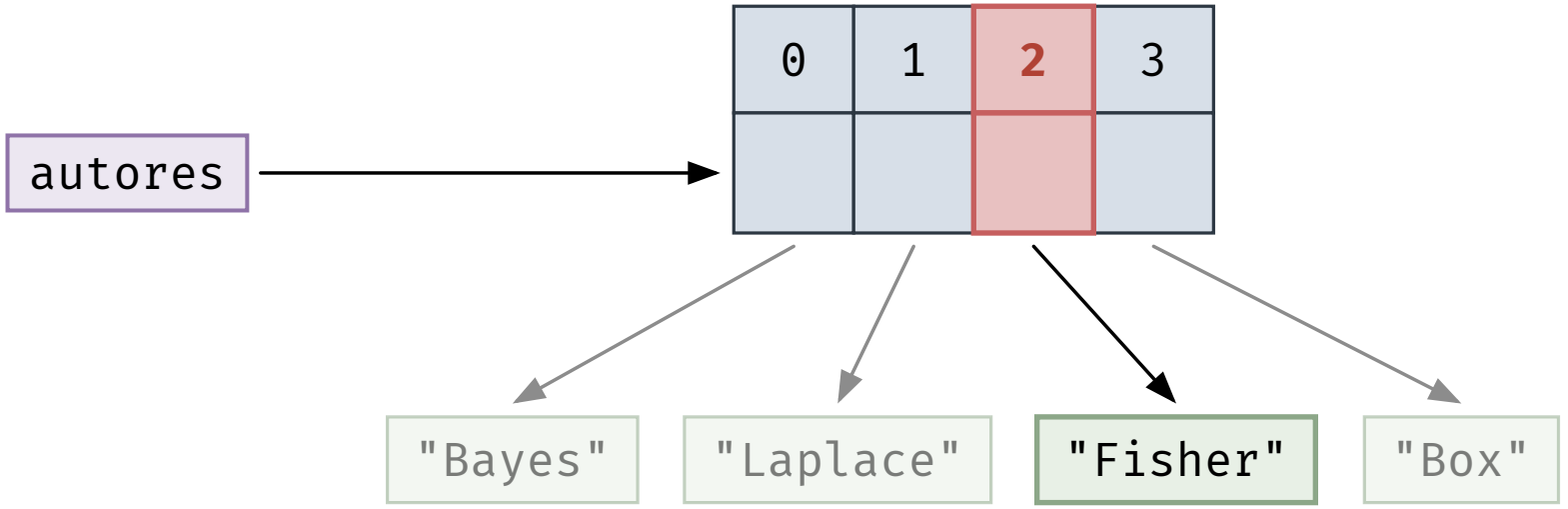
Selección de elementos

```
>>> autores[2]
```



Selección de elementos

```
>>> autores[2]
```




```
'Fisher'
```

Selección con índices negativos

Esto también funciona

```
>>> autores = autores = ["Bayes", "Laplace", "Fisher", "Box"]
>>> autores[-1]
```

Selección con índices negativos

Esto también funciona

```
>>> autores = autores = ["Bayes", "Laplace", "Fisher", "Box"]
>>> autores[-1]
'Box'
```

Selección con índices negativos

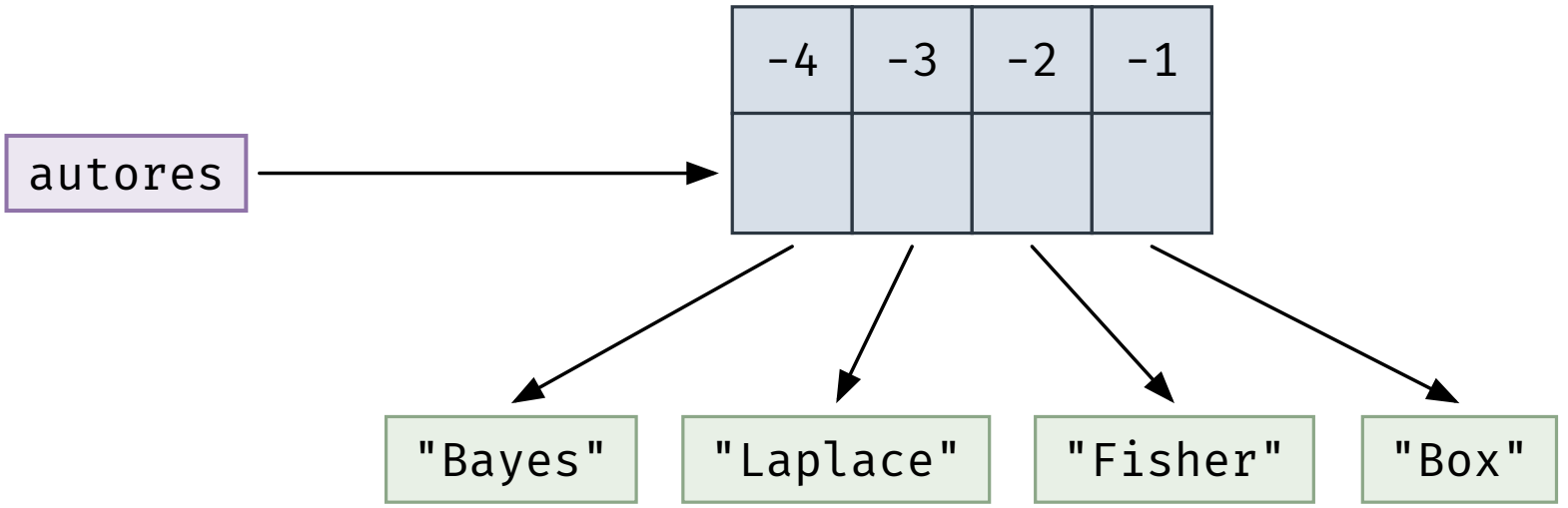
Esto también funciona

```
>>> autores = autores = ["Bayes", "Laplace", "Fisher", "Box"]
>>> autores[-1]
'Box'
```

En general

- El índice `-1` indica el último elemento.
- El índice `-2` indica el penúltimo elemento.
- Y así sucesivamente.

Selección con índices negativos

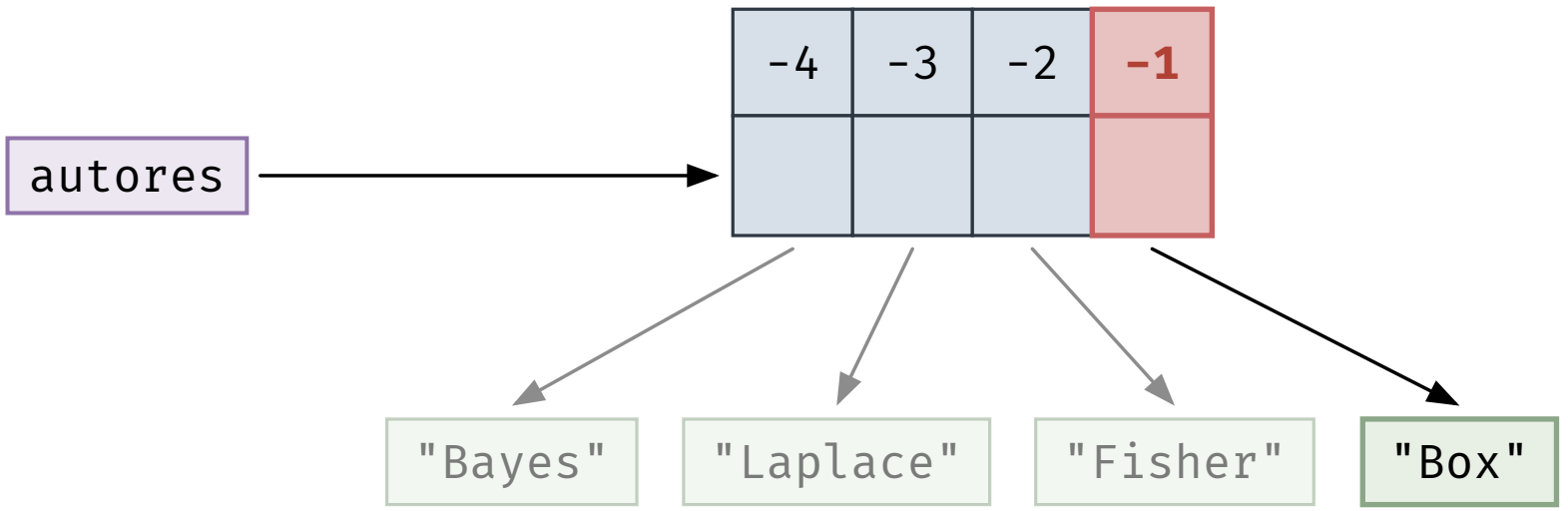


Selección con índices negativos

```
>>> autores[-1]
```

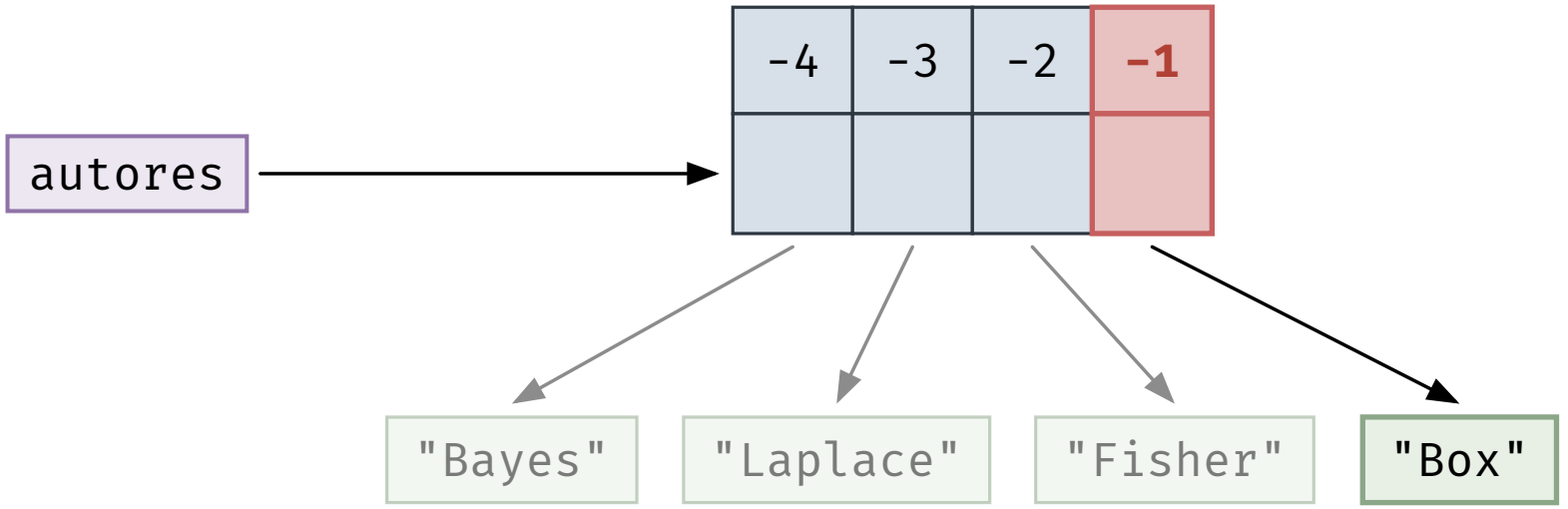
Selección con índices negativos

```
>>> autores[-1]
```



Selección con índices negativos

```
>>> autores[-1]
```



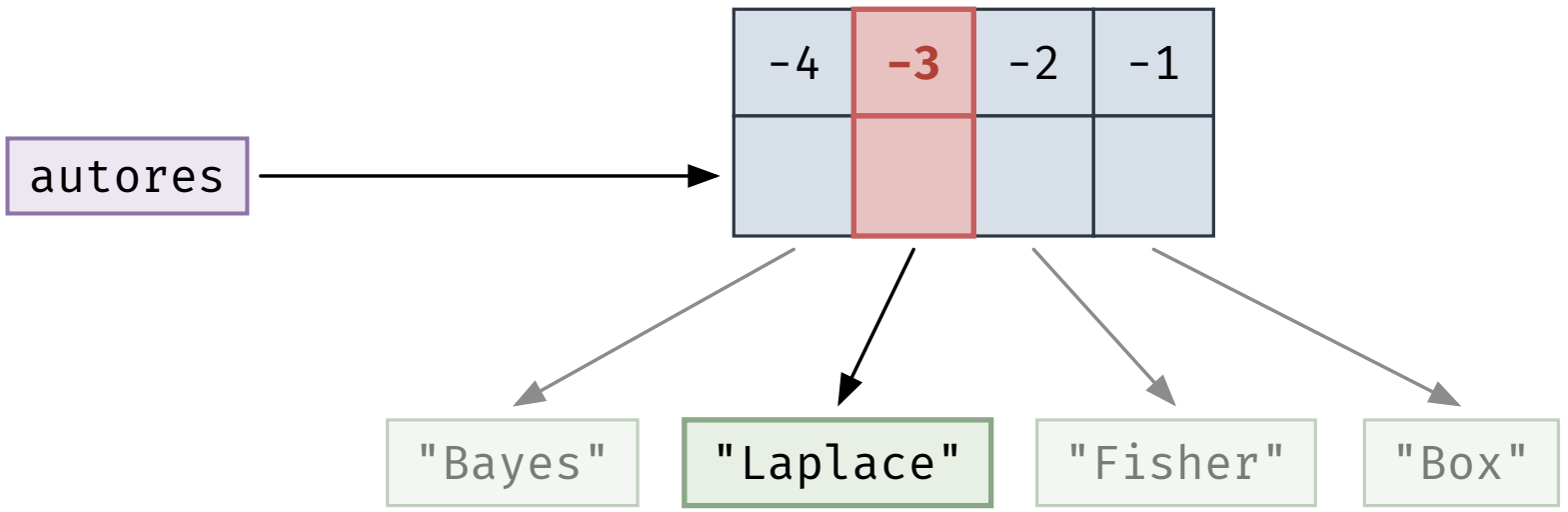

```
'Box'
```

Selección con índices negativos

```
>>> autores[-3]
```

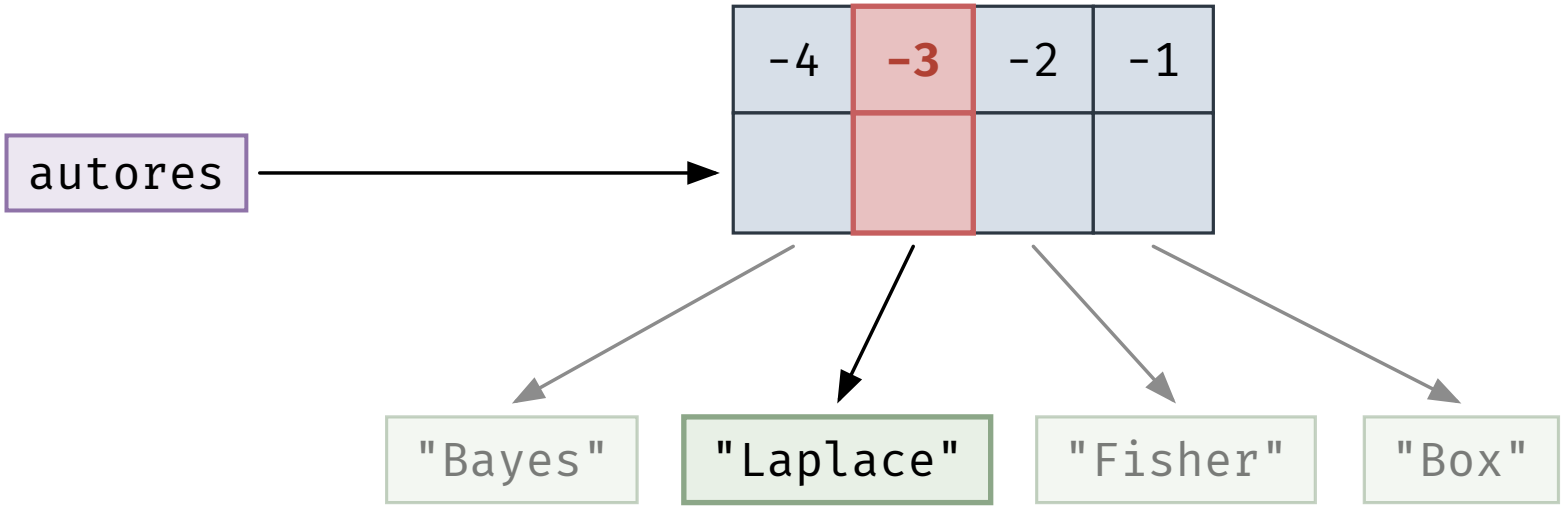
Selección con índices negativos

```
>>> autores[-3]
```



Selección con índices negativos

```
>>> autores[-3]
```




```
'Laplace'
```

Selección de sublistas

Se usan rebanadas o *slices* de la siguiente manera:

```
>>> lista[inicio:fin]
```

Selección de sublistas

Se usan rebanadas o *slices* de la siguiente manera:

```
>>> lista[inicio:fin]
```

Por ejemplo:

```
>>> ingredientes = ["Azúcar", "Flores", "Colores", "Sustancia X"]
>>> ingredientes[1:3]
```

Selección de sublistas

Se usan rebanadas o *slices* de la siguiente manera:

```
>>> lista[inicio:fin]
```

Por ejemplo:

```
>>> ingredientes = ["Azúcar", "Flores", "Colores", "Sustancia X"]
>>> ingredientes[1:3]
['Flores', 'Colores']
```

Selección de sublistas

Se usan rebanadas o *slices* de la siguiente manera:

```
>>> lista[inicio:fin]
```

Por ejemplo:

```
>>> ingredientes = ["Azúcar", "Flores", "Colores", "Sustancia X"]
>>> ingredientes[1:3]
['Flores', 'Colores']
```

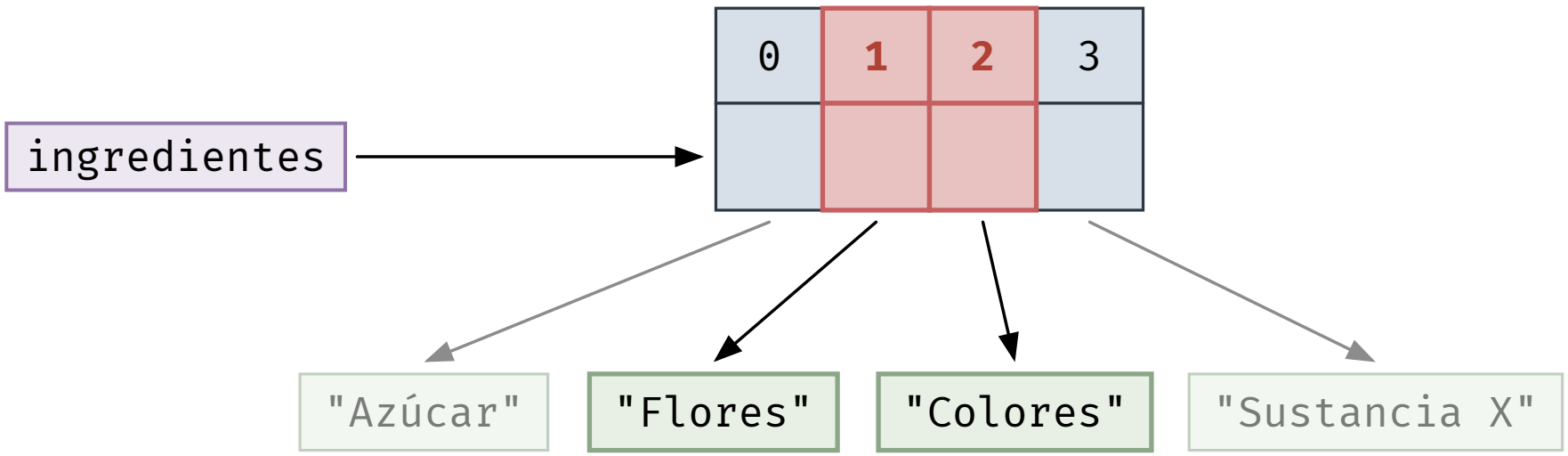
- Se incluye el límite inferior
- Se excluye el límite superior
- `[inicio, fin)`

Selección de sublistas

```
>>> ingredientes[1:3]
```

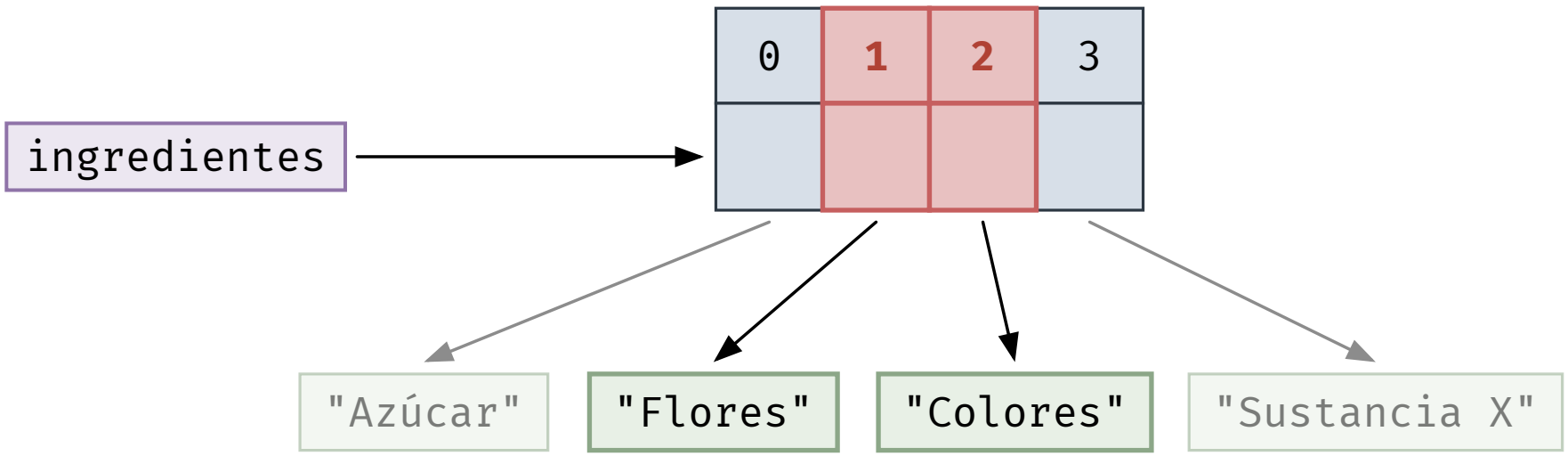
Selección de sublistas

```
>>> ingredientes[1:3]
```



Selección de sublistas

```
>>> ingredientes[1:3]
```




```
['Flores', 'Colores']
```

Rebanadas con inicio implícito

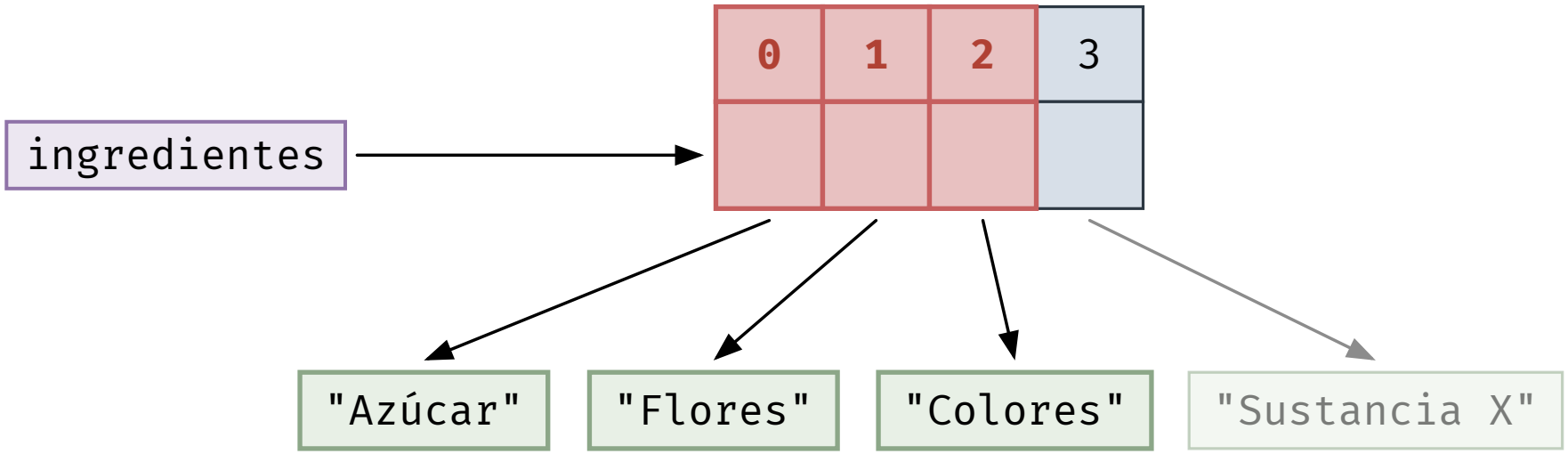
```
>>> lista[:fin]
```

Rebanadas con inicio implícito

```
>>> ingredientes[:3]
```

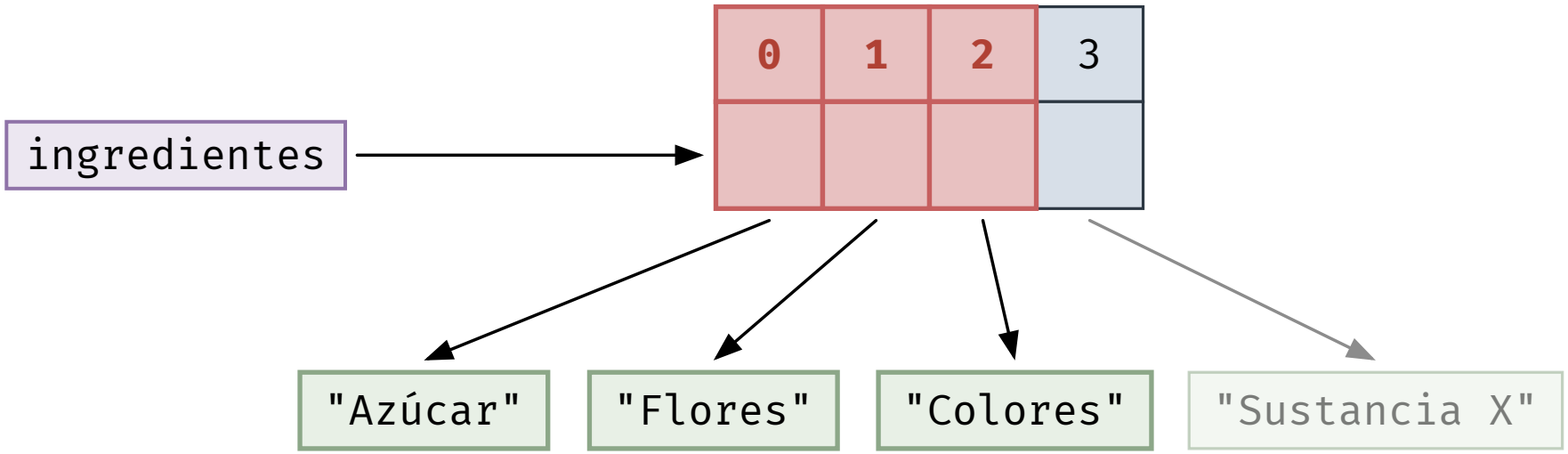
Rebanadas con inicio implícito

```
>>> ingredientes[:3]
```



Rebanadas con inicio implícito

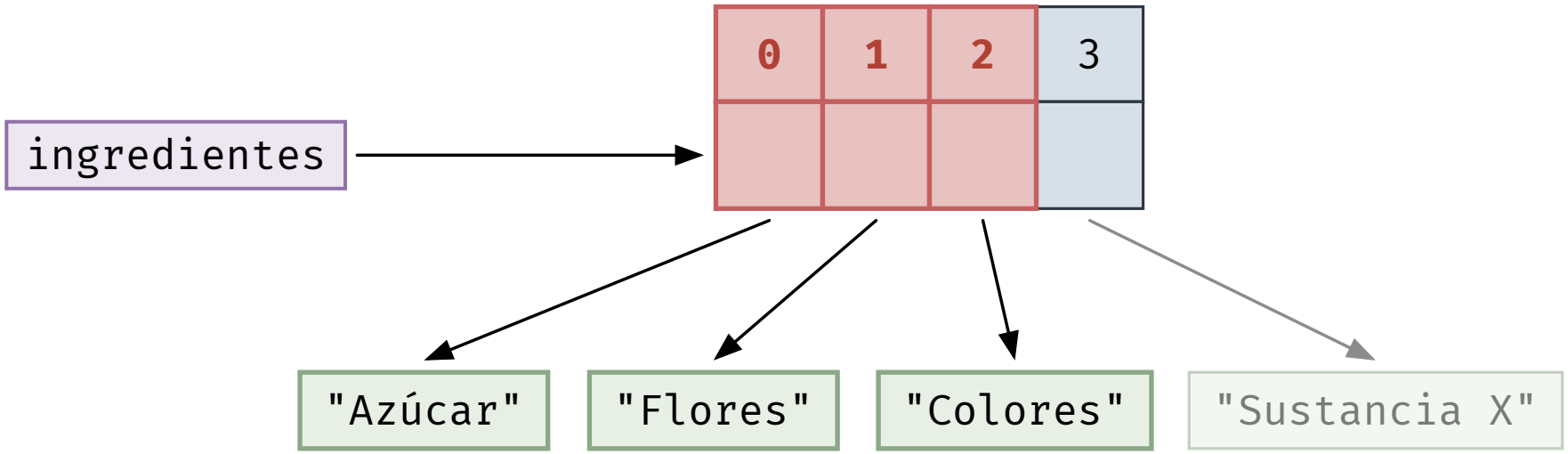
```
>>> ingredientes[:3]
```




```
['Azúcar', 'Flores', 'Colores']
```

Rebanadas con inicio implícito

```
>>> ingredientes[:3]
```




```
['Azúcar', 'Flores', 'Colores']
```

Un inicio implícito indica que se selecciona desde el principio

Rebanadas con fin implícito

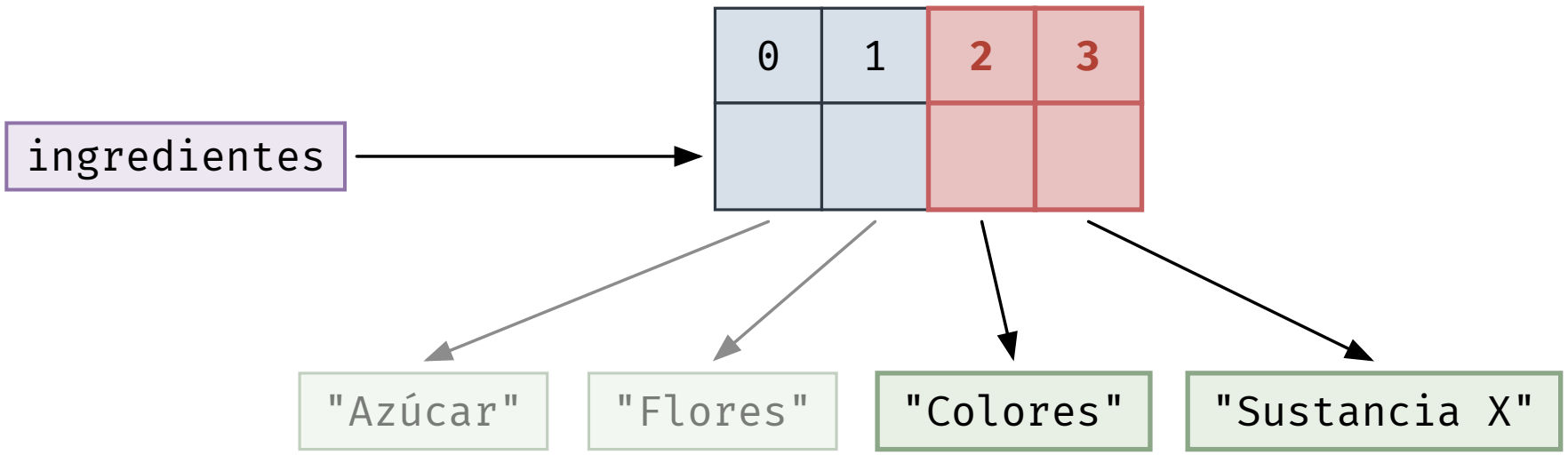
```
>>> lista[inicio:]
```

Rebanadas con fin implícito

```
>>> ingredientes[2:]
```

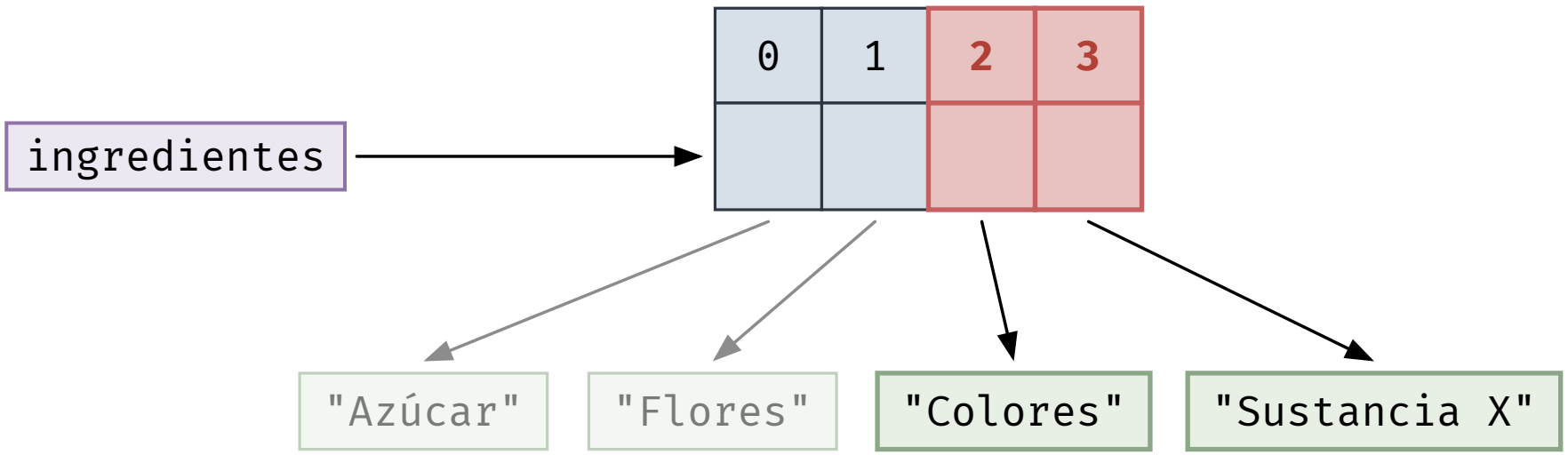
Rebanadas con fin implícito

```
>>> ingredientes[2:]
```



Rebanadas con fin implícito

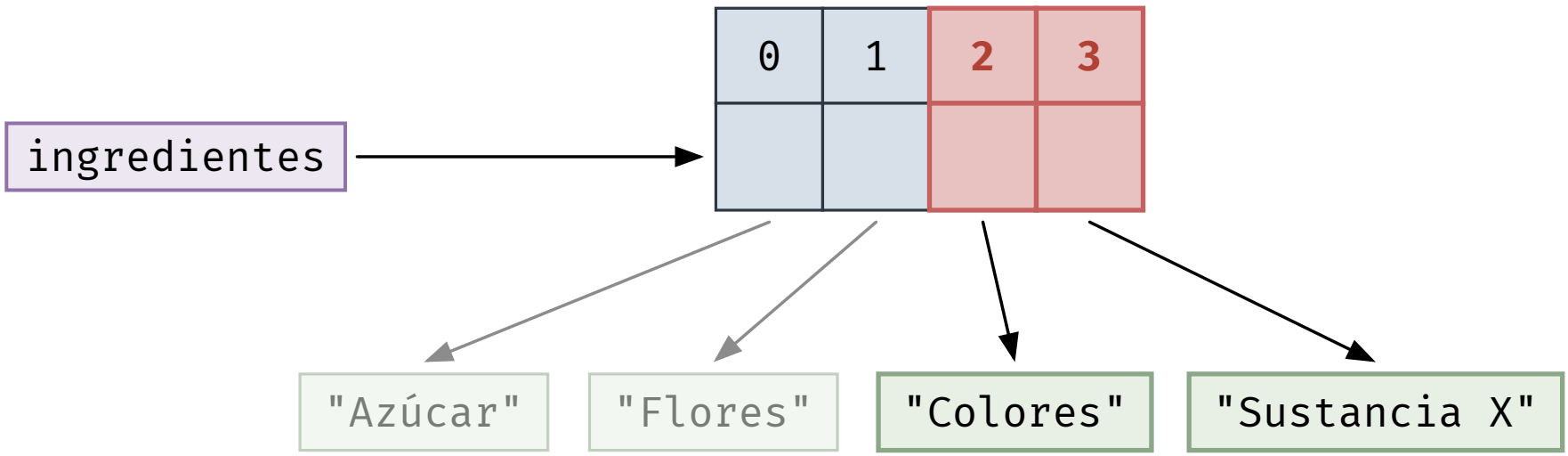
```
>>> ingredientes[2:]
```




```
['Colores', 'Sustancia X']
```

Rebanadas con fin implícito

```
>>> ingredientes[2:]
```




```
['Colores', 'Sustancia X']
```

Un fin implícito indica que se selecciona hasta el final

Rebanadas implícitas negativas

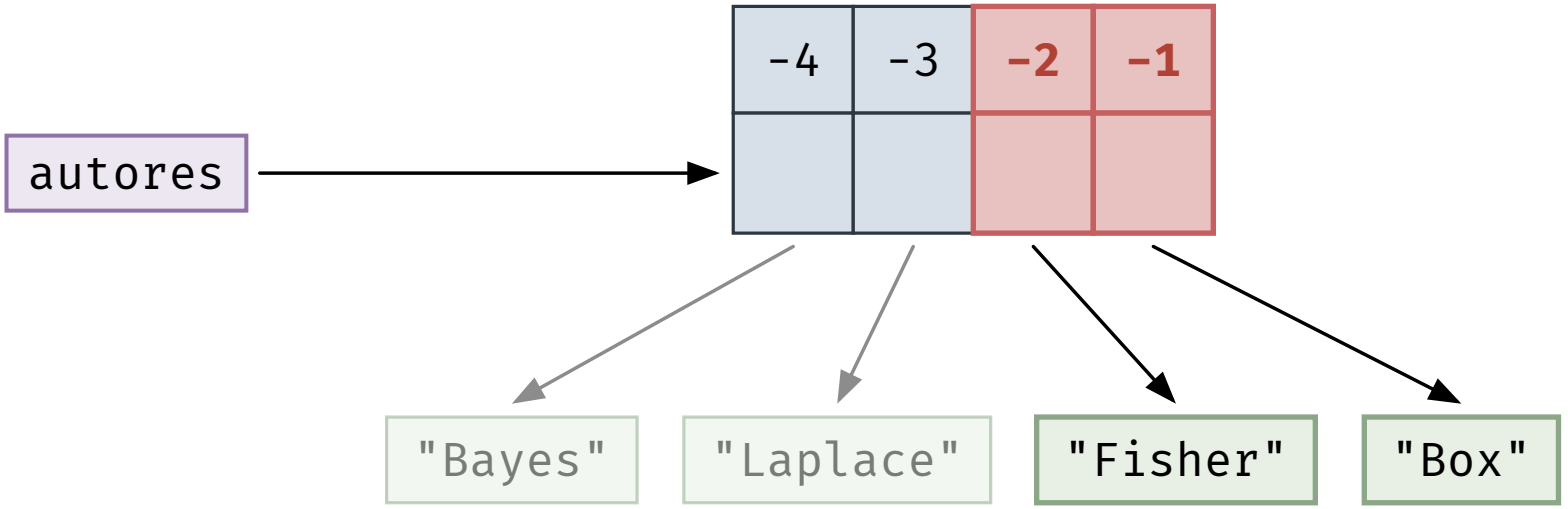
Rebanadas implícitas negativas 🤯

Rebanadas implícitas negativas 🤯

```
>>> autores[-2:]
```

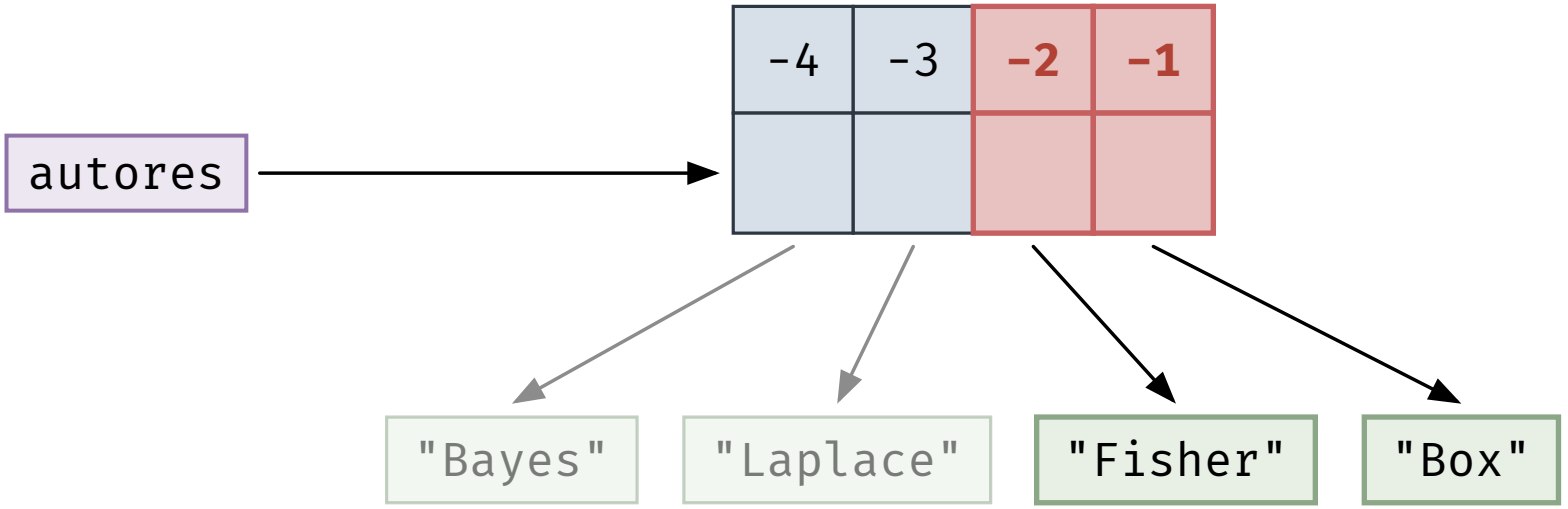
Rebanadas implícitas negativas 🤯

```
>>> autores[-2:]
```



Rebanadas implícitas negativas 🤯

```
>>> autores[-2:]
```




```
['Fisher', 'Box']
```

Selección de sublistas

i ¿Sabías qué?

Es válido no pasar ni `inicio` ni `fin` al crear un *slice*.

```
>>> lista[:]
```

El siguiente bloque devuelve una **copia** de la lista original:

```
>>> ingredientes[:]
['Azúcar', 'Flores', 'Colores', 'Sustancia X']
```

¿Cómo podríamos comprobarlo? 🤔

Quiz

Quiz

Explique el resultado de las siguientes expresiones:

```
>>> [1, 2, 3] == [1.0, 2, 3.0]
```

Quiz

Explique el resultado de las siguientes expresiones:

```
>>> [1, 2, 3] == [1.0, 2, 3.0]
```

```
>>> cosas = ["a", "b", "c", "d", "e", "f"]
>>> cosas[:-2]
>>> cosas[-5:]
```

Quiz

Explique el resultado de las siguientes expresiones:

```
>>> [1, 2, 3] == [1.0, 2, 3.0]
```

```
>>> cosas = ["a", "b", "c", "d", "e", "f"]
>>> cosas[:-2]
>>> cosas[-5:]
```

```
>>> [0][0]
>>> [1][1]
```

Quiz

Explique el resultado de las siguientes expresiones:

```
>>> [1, 2, 3] == [1.0, 2, 3.0]
```

```
>>> cosas = ["a", "b", "c", "d", "e", "f"]
>>> cosas[:-2]
>>> cosas[-5:]
```

```
>>> [0][0]
>>> [1][1]
```

```
>>> cosas = [True, False, False, None]
>>> cosas[4 / 2]
```

Reemplazar elementos

```
>>> marcas = ["Puerto Blest", "Martínez", "Fuego Tostadores"]
```

Reemplazar elementos

```
>>> marcas = ["Puerto Blest", "Martínez", "Fuego Tostadores"]
```

Reemplazar un único elemento:

```
>>> marcas[0] = "Bialetti"
>>> marcas
['Bialetti', 'Martínez', 'Fuego Tostadores']
```

Reemplazar elementos

```
>>> marcas = ["Puerto Blest", "Martínez", "Fuego Tostadores"]
```

Reemplazar un único elemento:

```
>>> marcas[0] = "Bialetti"
>>> marcas
['Bialetti', 'Martínez', 'Fuego Tostadores']
```

Reemplazar una sublista:

```
>>> marcas[1:3] = ["Indios Verdes", "John & Joe"]
>>> marcas
['Bialetti', 'Indios Verdes', 'John & Joe']
```

Reemplazar elementos

¡Valen índices negativos!

```
>>> marcas[-3] = "Cuervo"
>>> marcas
['Cuervo', 'Indios Verdes', 'John & Joe']
```

Reemplazar elementos

¡Valen índices negativos!

```
>>> marcas[-3] = "Cuervo"
>>> marcas
['Cuervo', 'Indios Verdes', 'John & Joe']
```

También en rebanadas:

```
>>> marcas[-2:] = ["Martínez", "Fuego Tostadores"]
>>> marcas
['Cuervo', 'Martínez', 'Fuego Tostadores']
```

Reemplazar elementos

¡Valen índices negativos!

```
>>> marcas[-3] = "Cuervo"
>>> marcas
['Cuervo', 'Indios Verdes', 'John & Joe']
```

También en rebanadas:

```
>>> marcas[-2:] = ["Martínez", "Fuego Tostadores"]
>>> marcas
['Cuervo', 'Martínez', 'Fuego Tostadores']
```

```
>>> marcas[-3:-1] = ["Puerto Blest", "Indigo"]
>>> marcas
['Puerto Blest', 'Indigo', 'Fuego Tostadores']
```

Agregar elementos, al final

De a uno, al final, utilizando el método `.append()`:

```
>>> vocales = ["a", "e", "i", "o"]
>>> vocales.append("u")
```

[1] En realidad, cualquier otra secuencia.

Agregar elementos, al final

De a uno, al final, utilizando el método `.append()`:

```
>>> vocales = ["a", "e", "i", "o"]
>>> vocales.append("u")
>>> vocales
['a', 'e', 'i', 'o', 'u']
```

[1] En realidad, cualquier otra secuencia.

Agregar elementos, al final

De a uno, al final, utilizando el método `.append()`:

```
>>> vocales = ["a", "e", "i", "o"]
>>> vocales.append("u")
>>> vocales
['a', 'e', 'i', 'o', 'u']
```

Con otra lista[1], utilizando `.extend()`:

```
>>> bartulos = [3.14, "casa", 11]
>>> bartulos.extend(["casa", True])
```

[1] En realidad, cualquier otra secuencia.

Agregar elementos, al final

De a uno, al final, utilizando el método `.append()`:

```
>>> vocales = ["a", "e", "i", "o"]
>>> vocales.append("u")
>>> vocales
['a', 'e', 'i', 'o', 'u']
```

Con otra lista[1], utilizando `.extend()`:

```
>>> bartulos = [3.14, "casa", 11]
>>> bartulos.extend(["casa", True])
>>> bartulos
[3.14, 'casa', 11, 'casa', True]
```

[1] En realidad, cualquier otra secuencia.

Agregar elementos, en posición arbitraria

También se puede insertar un elemento en una posición arbitraria con `.insert()`:

```
>>> logicos = [True, True, False, False]
>>> logicos.insert(2, "surprise")
>>> logicos
```

Agregar elementos, en posición arbitraria

También se puede insertar un elemento en una posición arbitraria con `.insert()`:

```
>>> logicos = [True, True, False, False]
>>> logicos.insert(2, "surprise")
>>> logicos
[True, True, 'surprise', False, False]
```

Agregar elementos, en posición arbitraria

También se puede insertar un elemento en una posición arbitraria con `.insert()`:

```
>>> logicos = [True, True, False, False]
>>> logicos.insert(2, "surprise")
>>> logicos
[True, True, 'surprise', False, False]
```

Importante

La posición indica dónde se agrega el nuevo elemento.
El resto de los elementos se desplaza hacia la derecha.

Agregar elementos, en posición arbitraria

También se puede insertar un elemento en una posición arbitraria con `.insert()`:

```
>>> logicos = [True, True, False, False]
>>> logicos.insert(2, "surprise")
>>> logicos
[True, True, 'surprise', False, False]
```

Importante

La posición indica dónde se agrega el nuevo elemento.
El resto de los elementos se desplaza hacia la derecha.

Tanto `.append()`, como `.extend()` e `insert()` son métodos ***in-place***.

¿*In-place*? ¿Qué es?

Definición

Un método *in-place* modifica el objeto original que realiza la llamada, no crea uno nuevo.

En Python, estos métodos suelen devolver un valor: `None`.

¿*In-place*? ¿Qué es?

Definición

Un método *in-place* modifica el objeto original que realiza la llamada, no crea uno nuevo.

En Python, estos métodos suelen devolver un valor: `None`.

Pregunta

¿Cómo podríamos verificar que el objeto al que apunta nuestra variable siga siendo el mismo luego de llamar al método *in-place*?

Eliminar elementos

El método `.pop()` elimina elementos a partir de su índice:

```
>>> autores = ["McElreath", "Jaynes", "Fisher"]
>>> autores.pop(1)
```

Eliminar elementos

El método `.pop()` elimina elementos a partir de su índice:

```
>>> autores = ["McElreath", "Jaynes", "Fisher"]
>>> autores.pop(1)
>>> autores
['McElreath', 'Fisher']
```

El método `.remove()` lo hace a partir del valor:

```
>>> galletitas = ["Melba", "Oreo", "Macucas", "Rumba"]
>>> galletitas.remove("Oreo")
```

Eliminar elementos

El método `.pop()` elimina elementos a partir de su índice:

```
>>> autores = ["McElreath", "Jaynes", "Fisher"]
>>> autores.pop(1)
>>> autores
['McElreath', 'Fisher']
```

El método `.remove()` lo hace a partir del valor:

```
>>> galletitas = ["Melba", "Oreo", "Macucas", "Rumba"]
>>> galletitas.remove("Oreo")
>>> galletitas
['Melba', 'Macucas', 'Rumba']
```

Eliminar elementos: detalles

El método `.pop()` **devuelve** el valor eliminado:

```
>>> l = [10, 100, 1000]
>>> e = l.pop(-1)
```

Eliminar elementos: detalles

El método `.pop()` **devuelve** el valor eliminado:

```
>>> l = [10, 100, 1000]
>>> e = l.pop(-1)
>>> e
1000
```

Si no se le pasa ningún valor, extrae por defecto el último:

```
>>> l = ["a", "bb", "ccc"]
```

Eliminar elementos: detalles

El método `.pop()` **devuelve** el valor eliminado:

```
>>> l = [10, 100, 1000]
>>> e = l.pop(-1)
>>> e
1000
```

Si no se le pasa ningún valor, extrae por defecto el último:

```
>>> l = ["a", "bb", "ccc"]
>>> l.pop()
'ccc'
```

Eliminar elementos: detalles

Por otro lado, `.remove()` no devuelve nada (¿qué significa?)

```
>>> numeros = [1, 2, 3, 4, 5]
>>> numeros.remove(3)
```

Eliminar elementos: detalles

Por otro lado, `.remove()` no devuelve nada (¿qué significa?)

```
>>> numeros = [1, 2, 3, 4, 5]
>>> numeros.remove(3)
```

Si el valor está repetido, elimina la primera ocurrencia:

```
>>> l = [1, 2, 3, 2, 2, 4]
>>> l.remove(2)
>>> l
```

Eliminar elementos: detalles

Por otro lado, `.remove()` no devuelve nada (¿qué significa?)

```
>>> numeros = [1, 2, 3, 4, 5]
>>> numeros.remove(3)
```

Si el valor está repetido, elimina la primera ocurrencia:

```
>>> l = [1, 2, 3, 2, 2, 4]
>>> l.remove(2)
>>> l
[1, 3, 2, 2, 4]
```

Eliminar elementos: detalles

Por otro lado, `.remove()` no devuelve nada (¿qué significa?)

```
>>> numeros = [1, 2, 3, 4, 5]
>>> numeros.remove(3)
```

Si el valor está repetido, elimina la primera ocurrencia:

```
>>> l = [1, 2, 3, 2, 2, 4]
>>> l.remove(2)
>>> l
[1, 3, 2, 2, 4]
```

Y si no existe, arroja un error.

Eliminar elementos con `del`

La sentencia `del`, que se usa para eliminar variables, también elimina elementos de una lista.

```
>>> galletitas = ["Melba", "Oreo", "Macucas", "Rumba"]
>>> del galletitas[3]
>>> galletitas
```

Eliminar elementos con `del`

La sentencia `del`, que se usa para eliminar variables, también elimina elementos de una lista.

```
>>> galletitas = ["Melba", "Oreo", "Macucas", "Rumba"]
>>> del galletitas[3]
>>> galletitas
['Melba', 'Oreo', 'Macucas']
```

Eliminar elementos con `del`

La sentencia `del`, que se usa para eliminar variables, también elimina elementos de una lista.

```
>>> galletitas = ["Melba", "Oreo", "Macucas", "Rumba"]
>>> del galletitas[3]
>>> galletitas
['Melba', 'Oreo', 'Macucas']
```

Al igual que `.pop()` y `remove()`, `del` también opera *in-place*.

Ordenar una lista

```
>>> galletitas = ["Melba", "Oreo", "Macucas", "Rumba"]
>>> galletitas.sort()
```

Ordenar una lista

```
>>> galletitas = ["Melba", "Oreo", "Macucas", "Rumba"]
>>> galletitas.sort()
>>> galletitas
['Macucas', 'Melba', 'Oreo', 'Rumba']
```

Ordenar una lista

```
>>> galletitas = ["Melba", "Oreo", "Macucas", "Rumba"]
>>> galletitas.sort()
>>> galletitas
['Macucas', 'Melba', 'Oreo', 'Rumba']
```

```
>>> galletitas = ["Melba", "Oreo", "Macucas", "Rumba"]
>>> galletitas.sort(reverse=True)
```

Ordenar una lista

```
>>> galletitas = ["Melba", "Oreo", "Macucas", "Rumba"]
>>> galletitas.sort()
>>> galletitas
['Macucas', 'Melba', 'Oreo', 'Rumba']
```

```
>>> galletitas = ["Melba", "Oreo", "Macucas", "Rumba"]
>>> galletitas.sort(reverse=True)
>>> galletitas
['Rumba', 'Oreo', 'Melba', 'Macucas']
```

Invertir una lista

```
>>> numeros = [1, 2, 3, 4, 5, 6]
>>> numeros.reverse()
```

Invertir una lista

```
>>> numeros = [1, 2, 3, 4, 5, 6]
>>> numeros.reverse()
>>> numeros
[6, 5, 4, 3, 2, 1]
```

¡¿Todo tiene que ser *in-place*?!

¡¿Todo tiene que ser *in-place*?!

Por suerte, no

Operaciones no destructivas sobre listas

```
>>> [1, 2, 3] + [4, 5, 6]
```

```
[1, 2, 3, 4, 5, 6]
```

Concatena listas y devuelve una nueva.

Operaciones no destructivas sobre listas

```
>>> [1, 2, 3] + [4, 5, 6]
>>> ["x", "y"] * 3
```

```
['x', 'y', 'x', 'y', 'x', 'y']
```

Repite la lista la cantidad de veces indicada.

Operaciones no destructivas sobre listas

```
>>> [1, 2, 3] + [4, 5, 6]
>>> ["x", "y"] * 3
>>> sorted([1, 10, 3, 2, 5, 6])
```

```
[1, 2, 3, 5, 6, 10]
```

Ordena y devuelve una nueva lista.

Operaciones no destructivas sobre listas

```
>>> [1, 2, 3] + [4, 5, 6]
>>> ["x", "y"] * 3
>>> sorted([1, 10, 3, 2, 5, 6])
>>> max([1, 2, 3])
```

```
3
```

Devuelve el mayor elemento.

Operaciones no destructivas sobre listas

```
>>> [1, 2, 3] + [4, 5, 6]
>>> ["x", "y"] * 3
>>> sorted([1, 10, 3, 2, 5, 6])
>>> max([1, 2, 3])
>>> min([1, 2, 3])
```

```
1
```

Devuelve el menor elemento.

Operaciones no destructivas sobre listas

```
>>> [1, 2, 3] + [4, 5, 6]
>>> ["x", "y"] * 3
>>> sorted([1, 10, 3, 2, 5, 6])
>>> max([1, 2, 3])
>>> min([1, 2, 3])
>>> sum([1, 2, 3])
```

```
6
```

Calcula la suma de los elementos.

Operaciones no destructivas sobre listas

```
>>> [1, 2, 3] + [4, 5, 6]
>>> ["x", "y"] * 3
>>> sorted([1, 10, 3, 2, 5, 6])
>>> max([1, 2, 3])
>>> min([1, 2, 3])
>>> sum([1, 2, 3])
>>> ["a", "b", "c"].index("b")
```

```
1
```

Devuelve la posición de la primera coincidencia.

Operaciones no destructivas sobre listas

```
>>> [1, 2, 3] + [4, 5, 6]
>>> ["x", "y"] * 3
>>> sorted([1, 10, 3, 2, 5, 6])
>>> max([1, 2, 3])
>>> min([1, 2, 3])
>>> sum([1, 2, 3])
>>> ["a", "b", "c"].index("b")
>>> [10, 10, 11, 12].count(10)
```

```
2
```

Cuenta cuántas veces aparece un valor.

Operaciones no destructivas sobre listas

```
>>> [1, 2, 3] + [4, 5, 6]
>>> ["x", "y"] * 3
>>> sorted([1, 10, 3, 2, 5, 6])
>>> max([1, 2, 3])
>>> min([1, 2, 3])
>>> sum([1, 2, 3])
>>> ["a", "b", "c"].index("b")
>>> [10, 10, 11, 12].count(10)
>>> len([1, 1, 1, 1])
```

```
4
```

Devuelve la cantidad de elementos.

Tuplas

Definición

Esto es una tupla

```
>>> (1, 2, 3, 4, 5)
(1, 2, 3, 4, 5)
```

Definición

Esto es una tupla

```
>>> (1, 2, 3, 4, 5)
(1, 2, 3, 4, 5)
```

Esto también

```
>>> "LE", "LC", "DNI"
('LE', 'LC', 'DNI')
```

Definición

Esto es una tupla

```
>>> (1, 2, 3, 4, 5)
(1, 2, 3, 4, 5)
```

Esto también

```
>>> "LE", "LC", "DNI"
('LE', 'LC', 'DNI')
```

Pero, ¿cómo se define?

Una tupla es...

- Secuencia ordenada de objetos

Definición

Esto es una tupla

```
>>> (1, 2, 3, 4, 5)
(1, 2, 3, 4, 5)
```

Esto también

```
>>> "LE", "LC", "DNI"
('LE', 'LC', 'DNI')
```

Pero, ¿cómo se define?

Una tupla es...

- Secuencia ordenada de objetos
- Inmutable

Creación

```
>>> tipos_identificacion = ("LE", "LC", "DNI", "CUIT", "CUIL")
```

Creación

```
>>> tipos_identificacion = ("LE", "LC", "DNI", "CUIT", "CUIL")
>>> tipos_identificacion
('LE', 'LC', 'DNI', 'CUIT', 'CUIL')
```

Creación

```
>>> tipos_identificacion = ("LE", "LC", "DNI", "CUIT", "CUIL")
>>> tipos_identificacion
('LE', 'LC', 'DNI', 'CUIT', 'CUIL')
>>> type(tipos_identificacion)
<class 'tuple'>
```

Creación

```
>>> tipos_identificacion = ("LE", "LC", "DNI", "CUIT", "CUIL")
>>> tipos_identificacion
('LE', 'LC', 'DNI', 'CUIT', 'CUIL')
>>> type(tipos_identificacion)
<class 'tuple'>
```

Acceder a elementos

```
>>> tipos_identificacion[0]
'LE'
```

Acceder a elementos

```
>>> tipos_identificacion[0]
'LE'
>>> tipos_identificacion[-1]
'CUIL'
```

Acceder a elementos

```
>>> tipos_identificacion[0]
'LE'
>>> tipos_identificacion[-1]
'CUIL'
>>> tipos_identificacion[2:4]
('DNI', 'CUIT')
```

Acceder a elementos

```
>>> tipos_identificacion[0]
'LE'
>>> tipos_identificacion[-1]
'CUIL'
>>> tipos_identificacion[2:4]
('DNI', 'CUIT')
```

El indexado y el *slicing* funcionan igual que en listas. Al seleccionar una porción, se obtiene una nueva tupla.

Inmutabilidad

Al contrario de las listas, no podemos reasignar elementos:

```
>>> tipos_identificacion[0] = "NUEVO"
TypeError: 'tuple' object does not support item assignment
```

Inmutabilidad

Al contrario de las listas, no podemos reasignar elementos:

```
>>> tipos_identificacion[0] = "NUEVO"
TypeError: 'tuple' object does not support item assignment
```

Tampoco existen métodos como `.append()`, `.remove()` o `.sort()`.

Operaciones útiles sobre tuplas

En general, las operaciones no destructivas que funcionan con listas también funcionan con tuplas.

```
>>> sorted((4, 1, 3, 2))
[1, 2, 3, 4]
```

Operaciones útiles sobre tuplas

En general, las operaciones no destructivas que funcionan con listas también funcionan con tuplas.

```
>>> sorted((4, 1, 3, 2))
[1, 2, 3, 4]
>>> min((4, 1, 3, 2))
1
```

Operaciones útiles sobre tuplas

En general, las operaciones no destructivas que funcionan con listas también funcionan con tuplas.

```
>>> sorted((4, 1, 3, 2))
[1, 2, 3, 4]
>>> min((4, 1, 3, 2))
1
>>> sum((4, 1, 3, 2))
10
```

Operaciones útiles sobre tuplas

En general, las operaciones no destructivas que funcionan con listas también funcionan con tuplas.

```
>>> sorted((4, 1, 3, 2))
[1, 2, 3, 4]
>>> min((4, 1, 3, 2))
1
>>> sum((4, 1, 3, 2))
10
>>> len((4, 1, 3, 2))
4
```

Operaciones útiles sobre tuplas

En general, las operaciones no destructivas que funcionan con listas también funcionan con tuplas.

```
>>> sorted((4, 1, 3, 2))
[1, 2, 3, 4]
>>> min((4, 1, 3, 2))
1
>>> sum((4, 1, 3, 2))
10
>>> len((4, 1, 3, 2))
4
>>> "DNI" in tipos_identificacion
True
```

Concatenar crea otro objeto

```
>>> original = ("LE", "LC", "DNI")
>>> nueva = original + ("CUIT",)
```

Concatenar crea otro objeto

```
>>> original = ("LE", "LC", "DNI")
>>> nueva = original + ("CUIT",)
>>> original is nueva
False
```

Concatenar crea otro objeto

```
>>> original = ("LE", "LC", "DNI")
>>> nueva = original + ("CUIT",)
>>> original is nueva
False
>>> original
('LE', 'LC', 'DNI')
```

Concatenar crea otro objeto

```
>>> original = ("LE", "LC", "DNI")
>>> nueva = original + ("CUIT",)
>>> original is nueva
False
>>> original
('LE', 'LC', 'DNI')
>>> nueva
('LE', 'LC', 'DNI', 'CUIT')
```

Tuplas como registros

Una tupla es útil para representar estructuras pequeñas que no deberían cambiar.

```
>>> personas = [("Juan", 29), ("Carla", 34), ("Evelina", 33)]
```

Tuplas como registros

Una tupla es útil para representar estructuras pequeñas que no deberían cambiar.

```
>>> personas = [("Juan", 29), ("Carla", 34), ("Evelina", 33)]
>>> personas[1]
('Carla', 34)
```

Tuplas como registros

Una tupla es útil para representar estructuras pequeñas que no deberían cambiar.

```
>>> personas = [("Juan", 29), ("Carla", 34), ("Evelina", 33)]
>>> personas[1]
('Carla', 34)
>>> personas[1][0]
'Carla'
```

Tuplas como registros

Una tupla es útil para representar estructuras pequeñas que no deberían cambiar.

```
>>> personas = [("Juan", 29), ("Carla", 34), ("Evelina", 33)]
>>> personas[1]
('Carla', 34)
>>> personas[1][0]
'Carla'
```

Cada tupla es un registro `(nombre, edad)`.

Lista vs. tupla

- `[]` crea listas; `()` suele usarse para tuplas.[1]
- Las listas son mutables; las tuplas son inmutables.
- Si la colección debe cambiar, usar `list`.
- Si la colección no debe cambiar, usar `tuple`.

[1] No siempre: en Python, la coma es lo que realmente define una tupla.

Detalle importante

En Python, la coma define la tupla:

```
>>> x = (3)
>>> type(x)
<class 'int'>
```

Detalle importante

En Python, la coma define la tupla:

```
>>> x = (3)
>>> type(x)
<class 'int'>
```

```
>>> y = (3,)
>>> type(y)
<class 'tuple'>
```

Diccionarios

Definición

Un diccionario es una estructura de datos que mapea claves con valores.

```
>>> personas = {"Juan": 29, "Carla": 34, "Evelina": 33, "Ana": 38}
>>> personas
{'Juan': 29, 'Carla': 34, 'Evelina': 33, 'Ana': 38}
```

Definición

Un diccionario es una estructura de datos que mapea claves con valores.

```
>>> personas = {"Juan": 29, "Carla": 34, "Evelina": 33, "Ana": 38}
>>> personas
{'Juan': 29, 'Carla': 34, 'Evelina': 33, 'Ana': 38}
```

```
>>> type(personas)
<class 'dict'>
```

Definición

Un diccionario es una estructura de datos que mapea claves con valores.

```
>>> personas = {"Juan": 29, "Carla": 34, "Evelina": 33, "Ana": 38}
>>> personas
{'Juan': 29, 'Carla': 34, 'Evelina': 33, 'Ana': 38}
```

```
>>> type(personas)
<class 'dict'>
```

En un diccionario, cada elemento es un par `clave: valor`.

¿Por qué usar diccionarios?

Queremos representar esta información:

```
>>> nombres = ["Juan", "Carla", "Evelina", "Ana"]
>>> edades = [29, 34, 33, 38]
```

¿Por qué usar diccionarios?

Queremos representar esta información:

```
>>> nombres = ["Juan", "Carla", "Evelina", "Ana"]
>>> edades = [29, 34, 33, 38]
```

Podríamos usar una lista de tuplas:

```
>>> personas = [
...     ("Juan", 29), ("Carla", 34), ("Evelina", 33), ("Ana", 38)
... ]
```

¿Por qué usar diccionarios?

Queremos representar esta información:

```
>>> nombres = ["Juan", "Carla", "Evelina", "Ana"]
>>> edades = [29, 34, 33, 38]
```

Podríamos usar una lista de tuplas:

```
>>> personas = [
...     ("Juan", 29), ("Carla", 34), ("Evelina", 33), ("Ana", 38)
... ]
```

Pero si queremos buscar la edad de una persona por nombre, un diccionario suele ser más natural.

Creación

Se definen entre llaves `{}`:

```
>>> precios = {"café": 3200, "tostado": 5800, "medialuna": 900}
```

También pueden crearse con `dict()`:

```
>>> descuentos = dict(lunes=0, martes=20, miercoles=10)
```

Creación

Se definen entre llaves `{}`:

```
>>> precios = {"café": 3200, "tostado": 5800, "medialuna": 900}
```

También pueden crearse con `dict()`:

```
>>> descuentos = dict(lunes=0, martes=20, miercoles=10)
>>> descuentos
{'lunes': 0, 'martes': 20, 'miercoles': 10}
```

Propiedades importantes

```
>>> d = {"nombre": "Juan", "edad": 29}
>>> len(d)
2
```

- `len(d)` cuenta pares `clave: valor`.

Propiedades importantes

```
>>> d = {"nombre": "Juan", "edad": 29}
>>> len(d)
2
```

- `len(d)` cuenta pares `clave: valor`.
- Las claves deben ser únicas.

Propiedades importantes

```
>>> d = {"nombre": "Juan", "edad": 29}
>>> len(d)
2
```

- `len(d)` cuenta pares `clave: valor`.
- Las claves deben ser únicas.
- Los valores pueden repetirse.

Propiedades importantes

```
>>> d = {"nombre": "Juan", "edad": 29}
>>> len(d)
2
```

- `len(d)` cuenta pares `clave: valor`.
- Las claves deben ser únicas.
- Los valores pueden repetirse.
- Es mutable.

Claves únicas

Si repetimos una clave, sobrevive la última asignación:

```
>>> d = {"a": 1, "b": 2, "a": 999}
```

Claves únicas

Si repetimos una clave, sobrevive la última asignación:

```
>>> d = {"a": 1, "b": 2, "a": 999}
>>> d
{'a': 999, 'b': 2}
```

Acceder a elementos

En diccionarios no usamos índices, usamos claves:

```
>>> personas = {"Juan": 29, "Carla": 34, "Evelina": 33, "Ana": 38}
```

Acceder a elementos

En diccionarios no usamos índices, usamos claves:

```
>>> personas = {"Juan": 29, "Carla": 34, "Evelina": 33, "Ana": 38}
>>> personas["Juan"]
29
```

Acceder a elementos

En diccionarios no usamos índices, usamos claves:

```
>>> personas = {"Juan": 29, "Carla": 34, "Evelina": 33, "Ana": 38}
>>> personas["Juan"]
29
>>> personas[0]
KeyError: 0
```

Verificar existencia

Los operadores `in` y `not in` evalúan la existencia de **claves**:

```
>>> d = {"color": "azul", "forma": "cuadrado"}
```

Verificar existencia

Los operadores `in` y `not in` evalúan la existencia de **claves**:

```
>>> d = {"color": "azul", "forma": "cuadrado"}
>>> "color" in d
True
```

Verificar existencia

Los operadores `in` y `not in` evalúan la existencia de **claves**:

```
>>> d = {"color": "azul", "forma": "cuadrado"}
>>> "color" in d
True
>>> "area" in d
False
```

Verificar existencia

Los operadores `in` y `not in` evalúan la existencia de **claves**:

```
>>> d = {"color": "azul", "forma": "cuadrado"}
>>> "color" in d
True
>>> "area" in d
False
>>> "area" not in d
True
```

Verificar existencia

Los operadores `in` y `not in` evalúan la existencia de **claves**:

```
>>> d = {"color": "azul", "forma": "cuadrado"}
>>> "color" in d
True
>>> "area" in d
False
>>> "area" not in d
True
>>> "azul" in d
False
```

Acceder a claves y valores

```
>>> d = {"color": "azul", "forma": "cuadrado"}
```

Acceder a claves y valores

```
>>> d = {"color": "azul", "forma": "cuadrado"}
>>> d.keys()
dict_keys(['color', 'forma'])
```

Acceder a claves y valores

```
>>> d = {"color": "azul", "forma": "cuadrado"}
>>> d.keys()
dict_keys(['color', 'forma'])
>>> d.values()
dict_values(['azul', 'cuadrado'])
```

Acceder a claves y valores

```
>>> d = {"color": "azul", "forma": "cuadrado"}
>>> d.keys()
dict_keys(['color', 'forma'])
>>> d.values()
dict_values(['azul', 'cuadrado'])
>>> d.items()
dict_items([('color', 'azul'), ('forma', 'cuadrado')])
```

Modificar y agregar elementos

La misma sintaxis sirve para ambas operaciones:

```
>>> personas = {"Juan": 29, "Carla": 34}
```

Modificar y agregar elementos

La misma sintaxis sirve para ambas operaciones:

```
>>> personas = {"Juan": 29, "Carla": 34}
>>> personas["Juan"] = 54   # Modifica registro
```

Modificar y agregar elementos

La misma sintaxis sirve para ambas operaciones:

```
>>> personas = {"Juan": 29, "Carla": 34}
>>> personas["Juan"] = 54   # Modifica registro
>>> personas["Marisa"] = 29 # Agrega registro
```

Modificar y agregar elementos

La misma sintaxis sirve para ambas operaciones:

```
>>> personas = {"Juan": 29, "Carla": 34}
>>> personas["Juan"] = 54   # Modifica registro
>>> personas["Marisa"] = 29 # Agrega registro
>>> personas
{'Juan': 54, 'Carla': 34, 'Marisa': 29}
```

Eliminar elementos

Con `del` eliminamos una clave sin recuperar el valor:

```
>>> descuentos = {"lunes": 0, "martes": 20, "miercoles": 10}
>>> del descuentos["martes"]
```

Eliminar elementos

Con `del` eliminamos una clave sin recuperar el valor:

```
>>> descuentos = {"lunes": 0, "martes": 20, "miercoles": 10}
>>> del descuentos["martes"]
>>> descuentos
{'lunes': 0, 'miercoles': 10}
```

Con `.pop()` eliminamos y recuperamos el valor:

```
>>> descuento_lunes = descuentos.pop("lunes")
```

Eliminar elementos

Con `del` eliminamos una clave sin recuperar el valor:

```
>>> descuentos = {"lunes": 0, "martes": 20, "miercoles": 10}
>>> del descuentos["martes"]
>>> descuentos
{'lunes': 0, 'miercoles': 10}
```

Con `.pop()` eliminamos y recuperamos el valor:

```
>>> descuento_lunes = descuentos.pop("lunes")
>>> descuento_lunes
0
```

Eliminar elementos

Con `del` eliminamos una clave sin recuperar el valor:

```
>>> descuentos = {"lunes": 0, "martes": 20, "miercoles": 10}
>>> del descuentos["martes"]
>>> descuentos
{'lunes': 0, 'miercoles': 10}
```

Con `.pop()` eliminamos y recuperamos el valor:

```
>>> descuento_lunes = descuentos.pop("lunes")
>>> descuento_lunes
0
>>> descuentos
{'miercoles': 10}
```

Actualizar diccionarios

`.update()` modifica el diccionario original:

```
>>> datos = {"nombre": "Guille", "ciudad": "Rosario"}
```

Actualizar diccionarios

`.update()` modifica el diccionario original:

```
>>> datos = {"nombre": "Guille", "ciudad": "Rosario"}
>>> datos_nuevos = {"ciudad": "Roldán", "hijos": 2}
>>> datos.update(datos_nuevos)
```

Actualizar diccionarios

`.update()` modifica el diccionario original:

```
>>> datos = {"nombre": "Guille", "ciudad": "Rosario"}
>>> datos_nuevos = {"ciudad": "Roldán", "hijos": 2}
>>> datos.update(datos_nuevos)
>>> datos
{'nombre': 'Guille', 'ciudad': 'Roldán', 'hijos': 2}
```

Actualizar diccionarios

`.update()` modifica el diccionario original:

```
>>> datos = {"nombre": "Guille", "ciudad": "Rosario"}
>>> datos_nuevos = {"ciudad": "Roldán", "hijos": 2}
>>> datos.update(datos_nuevos)
>>> datos
{'nombre': 'Guille', 'ciudad': 'Roldán', 'hijos': 2}
```

El operador `|` devuelve uno nuevo:

```
>>> d1 = {"a": 1, "b": 2}
>>> d2 = {"b": 10, "c": 25}
>>> d3 = d1 | d2
```

Actualizar diccionarios

`.update()` modifica el diccionario original:

```
>>> datos = {"nombre": "Guille", "ciudad": "Rosario"}
>>> datos_nuevos = {"ciudad": "Roldán", "hijos": 2}
>>> datos.update(datos_nuevos)
>>> datos
{'nombre': 'Guille', 'ciudad': 'Roldán', 'hijos': 2}
```

El operador `|` devuelve uno nuevo:

```
>>> d1 = {"a": 1, "b": 2}
>>> d2 = {"b": 10, "c": 25}
>>> d3 = d1 | d2
>>> d3
{'a': 1, 'b': 10, 'c': 25}
```

Estructuras anidadas

Los valores de un diccionario pueden ser otros diccionarios o listas:

```
>>> usuarios = {
...     "aeinstein": {
...         "nombre": "albert",
...         "apellido": "einstein",
...         "ciudad": "princeton",
...     },
...     "mcurie": {
...         "nombre": "marie",
...         "apellido": "curie",
...         "ciudad": "paris",
...     },
... }
>>> usuarios["aeinstein"]["ciudad"]
'princeton'
```

Listas, tuplas y diccionarios

- Las listas y tuplas son secuencias: se accede por posición.
  - Listas: se pueden modificar
  - Tuplas: no se pueden modificar
- Los diccionarios no son secuencias: se accede por clave.
- Si el problema es clave → valor, usar `dict`.