### 5.2.1. Cosas que, sin darnos cuenta, ya sabemos sobre las listas

Una ventaja de Python es que proporciona operadores y funciones similares para trabajar con tipos de datos similares. Las cadenas y las listas tienen algo en común: ambas son *secuencias* de datos, así pues, muchos de los operadores y funciones que trabajan sobre cadenas también lo hacen sobre listas. Por ejemplo, la función *len*, aplicada sobre una lista, nos dice cuántos elementos la integran:

```
>>> a = [1, 2, 3] ↲
>>> len(a) ↲
3
>>> len([0, 1, 10, 5]) ↲
4
>>> len([10]) ↲
1
```

La longitud de la lista vacía es 0:

```
>>> len([]) ↲
0
```

El operador + concatena listas:

```
>>> [1, 2] + [3, 4] ↲
[1, 2, 3, 4]
>>> a = [1, 2, 3] ↲
>>> [10, 20] + a ↲
[10, 20, 1, 2, 3]
```

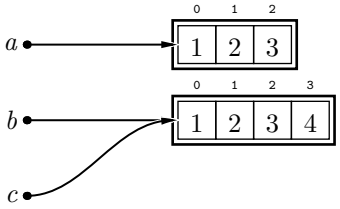
y el operador * repite un número dado de veces una lista:

```
>>> [1, 2] * 3 ↲
[1, 2, 1, 2, 1, 2]
>>> a = [1, 2, 3] ↲
>>> b = [10, 20] + a * 2 ↲
>>> b ↲
[10, 20, 1, 2, 3, 1, 2, 3]
```

Has de tener en cuenta que tanto + como * generan nuevas listas, sin modificar las originales. Observa este ejemplo:

```
>>> a = [1, 2, 3] ↲
>>> b = a + [4] ↲
>>> c = b ↲
```

La memoria queda así:




¿Ves? La asignación a *b* deja intacta la lista *a* porque apunta al resultado de concatenar algo a *a*. La operación de concatenación no modifica la lista original: reserva memoria para una nueva lista con tantos elementos como resultan de sumar la longitud de las listas concatenadas y, a continuación, copia los elementos de la primera lista seguidos por los

de la segunda lista en la nueva zona de memoria. Como asignamos a *b* el resultado de la concatenación, tenemos que *b* apunta a la lista recién creada. La tercera sentencia es una simple asignación a *c*, así que Python se limita a copiar la referencia.

El operador de indexación también es aplicable a las listas:

```
>>> a = [1, 2, 3] ↲
>>> a[1] ↲
2
>>> a[len(a)-1] ↲
3
>>> a[-1] ↲
3
```

A veces, el operador de indexación puede dar lugar a expresiones algo confusas a primera vista:

```
>>> [1, 2, 3][0] ↲
1
```

En este ejemplo, el primer par de corchetes indica el principio y final de la lista (formada por el 1, el 2 y el 3) y el segundo par indica el índice del elemento al que deseamos acceder (el primero, es decir, el de índice 0).

EJERCICIOS

▶ 214 ¿Qué aparecerá por pantalla al evaluar la expresión `[1][0]`? ¿Y al evaluar la expresión `[][0]`?

De todos modos, no te preocupes por esa notación un tanto confusa: lo normal es que accedas a los elementos de listas que están almacenadas en variables, con lo que rara vez tendrás dudas.

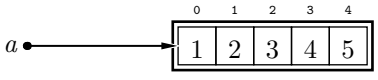
```
>>> a = [1, 2, 3] ↲
>>> a[0] ↲
1
```

También el operador de corte es aplicable a las listas:

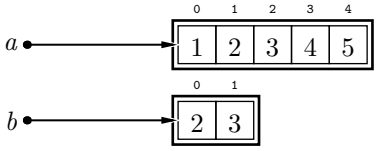
```
>>> a = [1, 2, 3] ↲
>>> a[1:-1] ↲
[2]
>>> a[1:] ↲
[2, 3]
```

Has de tener en cuenta que un corte siempre se extrae copiando un fragmento de la lista, por lo que comporta la reserva de memoria para crear una nueva lista. Analiza la siguiente secuencia de acciones y sus efectos sobre la memoria:

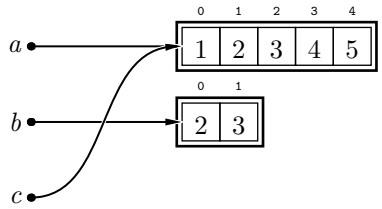
```
>>> a = [1, 2, 3, 4, 5] ↲
```



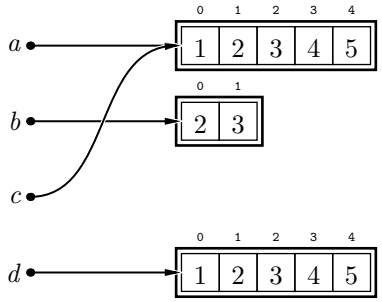

```
>>> b = a[1:3] ↲
```



```
>>> c = a ↲
```




```
>>> d = a[:] ↲
```




Si deseas asegurarte de que trabajas con una copia de una lista y no con la misma lista (a través de una referencia) utiliza el operador de corte en la asignación.

EJERCICIOS

▶ 215 Hemos asignado a *x* la lista `[1, 2, 3]` y ahora queremos asignar a *y* una copia. Podríamos hacer *y* = *x*`[:]`, pero parece que *y* = *x* + `[]` también funciona. ¿Es así? ¿Por qué?

El iterador **for-in** también recorre los elementos de una lista:

```
>>> for i in [1, 2, 3]: ↲
...     print i ↲
... ↲
1
2
3
```

De hecho, ya hemos utilizado bucles que iteran sobre listas. Cuando utilizamos un bucle **for-in** del modo convencional, es decir, haciendo uso de *range*, estamos recorriendo una lista:

```
>>> for i in range(1, 4): ↲
...     print i ↲
... ↲
1
2
3
```

Y es que *range*`(1, 4)` construye y devuelve la lista `[1, 2, 3]`:

```
>>> a = range(1, 4) ↲
>>> print a ↲
[1, 2, 3]
```

Una forma corriente de construir listas que contienen réplicas de un mismo valor se ayuda del operador ∗. Supongamos que necesitamos una lista de 10 elementos, todos los cuales valen 0. Podemos hacerlo así:

```
>>> [0] * 10 ↲
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

EJERCICIOS

▶ 216 ¿Qué aparecerá por pantalla al ejecutar este programa?

```
print 'Principio'
for i in []:
    print 'paso', i
print 'y fin'
```

▶ 217 ¿Qué aparecerá por pantalla al ejecutar este programa?

```
for i in [1] * 10:
    print i
```