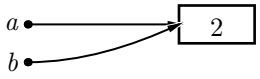
5.2.5. Mutabilidad, inmutabilidad y representación de la información en memoria

Python procura no consumir más memoria que la necesaria. Ciertos objetos son inmutables, es decir, no pueden modificar su valor. El número 2 es siempre el número 2. Es un objeto inmutable. Python procura almacenar en memoria una sola vez cada valor inmutable. Si dos o más variables contienen ese valor, sus referencias apuntan a la misma zona de memoria. Considera este ejemplo:

```
>>> a = 1 + 1 ↲
>>> b = 2 * 1 ↲
```

La memoria presenta, tras esas asignaciones, este aspecto:

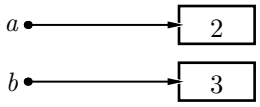


¿Y qué ocurre cuando modificamos el valor de una variable inmutable? No se modifica el contenido de la caja que contiene el valor, sino que el correspondiente puntero pasa a apuntar a una caja con el nuevo valor; y si ésta no existe, se crea.

Si a las asignaciones anteriores le siguen éstas:

```
>>> b = b + 1 ↲
```

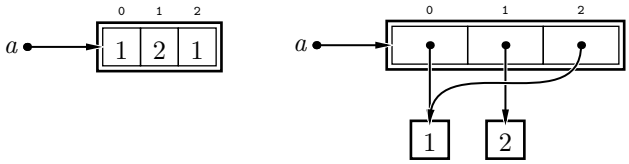
la memoria pasa a tener este aspecto:



También las cadenas Python son objetos inmutables[2]. Que lo sean tiene efectos sobre las operaciones que podemos efectuar con ellas. La asignación a un elemento de una cadena, por ejemplo está prohibida, así que Python la señala con un «error de tipo» (*TypeError*):

```
>>> a = 'Hola' ↲
>>> a[0] = 'h' ↲
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

Las listas se comportan de forma diferente: a diferencia de las cadenas, son mutables. De momento te hemos proporcionado una representación de las listas excesivamente simplificada. Hemos representando el resultado de la asignación `a = [1, 2, 1]` como se muestra a la izquierda, cuando lo correcto sería hacerlo como se muestra a la derecha:

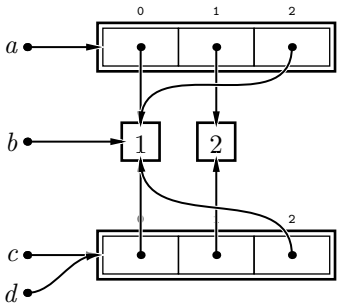


La realidad, como ves, es algo complicada: la lista almacena referencias a los valores, y no los propios valores. Pero aún no lo has visto todo. ¿Qué ocurre tras ejecutar estas sentencias?

```
>>> a = [1, 2, 1] ↲
>>> b = 1 ↲
>>> c = [1, 2, 1] ↲
>>> d = c ↲
```

Nada menos que esto:

[2]Aunque los ejemplos que hemos presentado con enteros no son directamente trasladables al caso de las cadenas. Aunque parezca paradójico, Python puede decidir por razones de eficiencia que dos cadenas con idéntico contenido se almacenen por duplicado.



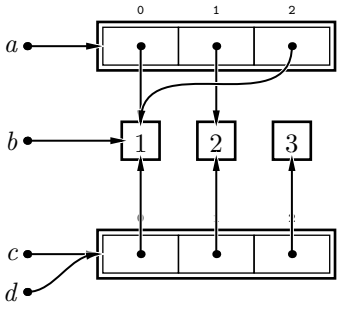
Como habrás observado, para cada aparición de un literal de lista, es decir, de una lista expresada explícitamente, (como `[1, 2, 1]`), Python ha reservado nueva memoria, aunque exista otra lista de idéntico valor. Así pues, *a* `= [1, 2, 1]` y *c* `= [1, 2, 1]` han generado sendas reservas de memoria y cada variable apunta a una zona de memoria diferente. Como el contenido de cada celda ha resultado ser un valor inmutable (un entero), se han compartido las referencias a los mismos. El operador **is** nos ayuda a confirmar nuestra hipótesis:

```
>>> a[0] is b ↲
True
>>> c[-1] is a[0] ↲
True
```

Modifiquemos ahora el contenido de una celda de una de las listas:

```
>>> d[2] = 3 ↲
```

El resultado es éste:

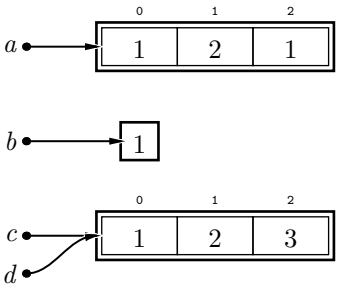


EJERCICIOS

▶ 228 Representa el estado de la memoria tras efectuar cada una de las siguientes asignaciones:

```
>>> a = [1, 2, 1] ↲
>>> b = 1 ↲
>>> c = [2, 1, 2] ↲
>>> d = c ↲
>>> d[2] = 3 ↲
>>> e = d[:1] ↲
>>> f = d[:] ↲
>>> f[0] = a[1] ↲
>>> f[1] = 1 ↲
```

Aunque los diagramas que hemos mostrado responden a la realidad, usaremos normalmente su versión simplificada (y, en cierto modo, «falsa»), pues es suficiente para el diseño de la mayor parte de programas que vamos a presentar. Con esta visión simplificada, la última figura se representaría así:


0
1
2
a
1
2
1
b
1
0
1
2
c
d
1
2
3