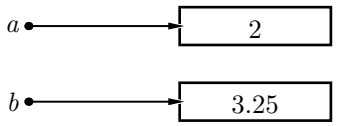
5.1.11. Referencias a cadenas

En el apartado 2.4 hemos representado las variables y su contenido con diagramas de cajas. Por ejemplo, las siguientes asignaciones:

```
>>> a = 2 ↲
>>> b = 3.25 ↲
```

conducen a una disposición de la información en la memoria que mostramos gráficamente así:

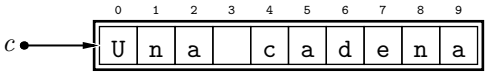


Decimos que *a apunta* al valor 2 y que *b apunta* al valor 3.25. La flecha recibe el nombre de puntero o referencia.

Con las cadenas representaremos los valores desglosando cada uno de sus caracteres en una caja individual con un índice asociado. El resultado de una asignación como ésta:

```
>>> c = 'Una cadena' ↲
```

se representará del siguiente modo:

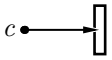


Decimos que la variable *c apunta* a la cadena `'Una cadena'`, que es una secuencia de caracteres.

La cadena vacía no ocupa ninguna celda de memoria y la representamos gráficamente de un modo especial. Una asignación como ésta:

```
>>> c = '' ↲
```

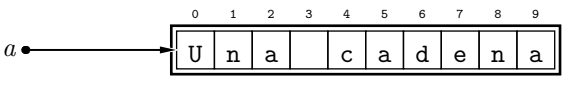
se representa así:



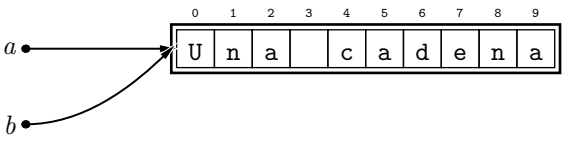
Que las variables contengan referencias a los datos y no los propios datos es muy útil para aprovechar la memoria del ordenador. El siguiente ejemplo te ilustrará el ahorro que se consigue.

```
>>> a = 'Una cadena' ↲
>>> b = a ↲
```

Tras ejecutar la primera acción tenemos:

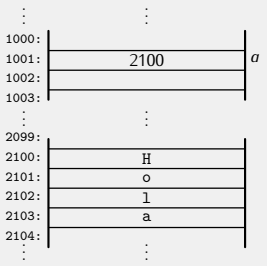


Y después de ejecutar la segunda:

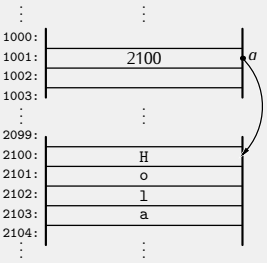


Las referencias son direcciones de memoria (I)

Vamos a darte una interpretación de las referencias que, aunque constituye una simplificación de la realidad, te permitirá entender qué son. Ya dijimos en el tema 1 que la memoria del computador se compone de una serie de celdas numeradas con sus direcciones. En cada celda cabe un escalar. La cadena `'Hola'` ocupa cuatro celdas, una por cada carácter. Por otra parte, una variable sólo puede contener un escalar. Como la dirección de memoria es un número y, por tanto, un escalar, el «truco» consiste en almacenar en la variable la dirección de memoria en la que empieza la cadena. Fíjate en este ejemplo en el que una variable ocupa la dirección de memoria 1001 y «contiene» la cadena `'Hola'`:



Como puedes ver, en realidad la cadena ocupa posiciones consecutivas a partir de una dirección determinada (en el ejemplo, la 2100) y la variable contiene el valor de dicha referencia. La flecha de los diagramas hace más «legibles» las referencias:



¡Tanto *a* como *b apuntan* a la misma cadena! Al asignar a una variable la cadena contenida en otra *únicamente se copia su referencia* y no cada uno de los caracteres que la componen. Si se hiciera del segundo modo, la memoria ocupada y el tiempo necesarios para la asignación serían tanto mayores cuanto más larga fuera la cadena. El método escogido únicamente copia el valor de la referencia, así que es independiente de la longitud de la cadena (y prácticamente instantáneo).

Has de tener en cuenta, pues, que una asignación únicamente altera el valor de un puntero. Pero otras operaciones con cadenas comportan la reserva de nueva memoria. Tomemos por caso el operador de concatenación. La concatenación toma dos cadenas y forma *una cadena nueva* que resulta de unir ambas, es decir, reserva memoria para una nueva cadena. Veamos paso a paso cómo funciona el proceso con un par de ejemplos. Fíjate en estas sentencias:

```
>>> a = 'otra␣' ↲
>>> b = 'cadena' ↲
>>> c = a + b ↲
```

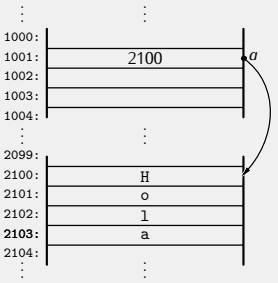
Podemos representar gráficamente el resultado de la ejecución de las dos primeras sentencias así:

Las referencias son direcciones de memoria (y II)

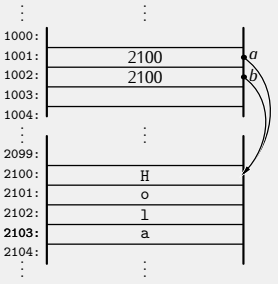
Veamos qué ocurre cuando dos variables comparten referencia. El ejemplo que hemos desarrollado en el texto estudia el efecto de estas dos asignaciones:

```
>>> a = 'Una cadena' ↲
>>> b = a ↲
```

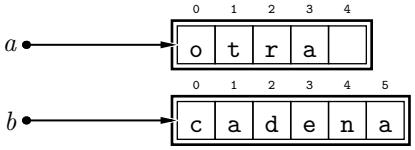
Como vimos antes, la primera asignación conduce a esta situación:



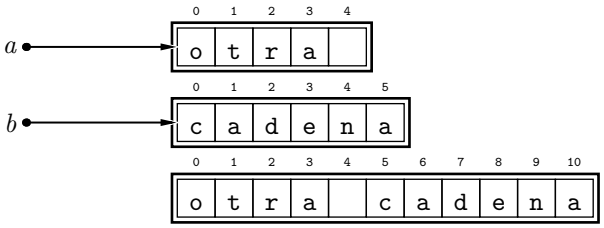
Pues bien, la segunda asignación copia en la dirección de *b* (que suponemos es la 1002) el valor que hay almacenado en la dirección de *a*, es decir, el valor 2100:



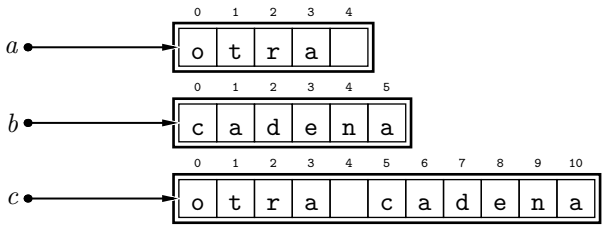
Copiar un valor escalar de una posición de memoria a otra es una acción muy rápida.



Analicemos ahora la tercera sentencia. En primer lugar, Python evalúa la expresión *a* + *b*, así que reserva un bloque de memoria con espacio para 11 caracteres y copia en ellos los caracteres de *a* seguidos de los caracteres de *b*:



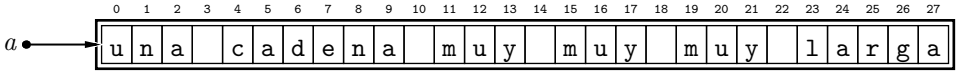
Y ahora que ha creado la nueva cadena, se ejecuta la asignación en sí, es decir, se hace que *c* apunte a la nueva cadena:



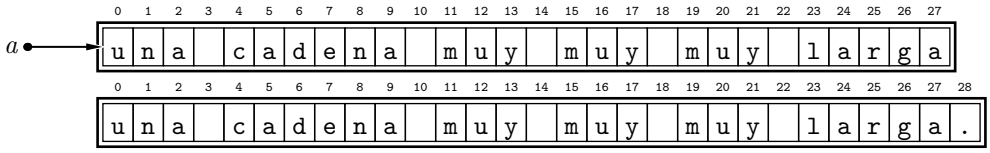
El orden en el que ocurren las cosas tiene importancia para entender cómo puede verse afectada la velocidad de ejecución de un programa por ciertas operaciones. Tomemos por caso estas dos órdenes:

```
>>> a = 'una cadena muy muy muy larga' ↲
>>> a = a + '.' ↲
```

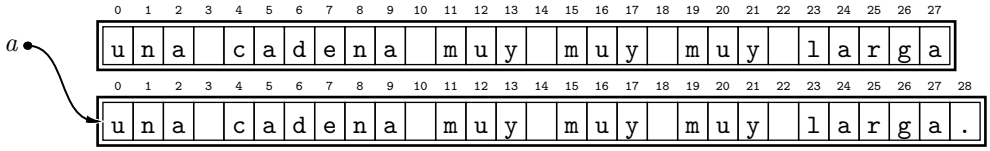
A simple vista parece que la primera sentencia será más lenta en ejecución que la segunda, pues comporta la reserva de una zona de memoria que puede ser grande (imagina si la cadena tuviera mil o incluso cien mil caracteres), mientras que la segunda sentencia se limita a añadir un solo carácter. Pero no es así: ambas tardan casi lo mismo. Veamos cuál es la razón. La primera sentencia reserva memoria para 28 caracteres, los guarda en ella y hace que *a* apunte a dicha zona:



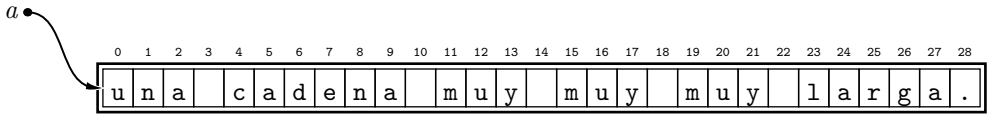
Y ahora veamos paso a paso qué ocurre al ejecutar la segunda sentencia. En primer lugar se evalúa la parte derecha, es decir, se reserva espacio para 29 caracteres y se copian en él los 28 caracteres de *a* y el carácter punto:



Y ahora, al ejecutar la asignación, la variable *a* pasa de apuntar a la zona de memoria original para apuntar a la nueva zona de memoria:



Como la zona inicial de memoria ya no se usa para nada, Python la «libera», es decir, considera que está disponible para futuras operaciones, con lo que, a efectos prácticos, desaparece:

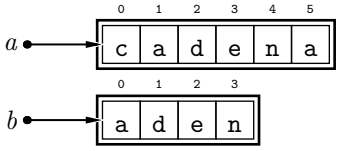


Como puedes ver, la sentencia que consiste en añadir un simple punto a una cadena es más costosa en tiempo que la que comporta una asignación a una variable de esa misma cadena.

El operador con asignación **+=** actúa exactamente igual con cadenas, así que sustituir la última sentencia por *a* **+= '.'** presenta el mismo problema.

El operador de corte también reserva una nueva zona de memoria:

```
>>> a = 'cadena' ↲
>>> b = a[1:-1] ↲
```



EJERCICIOS

▶ 212 Dibuja un diagrama con el estado de la memoria tras ejecutar estas sentencias:

```
>>> a = 'cadena' ↲
>>> b = a[2:3] ↲
>>> c = b + '' ↲
```

▶ 213 Dibuja diagramas que muestren el estado de la memoria paso a paso para esta secuencia de asignaciones.

```
>>> a = 'ab' ↲
>>> a *= 3 ↲
>>> b = a ↲
>>> c = a[:] ↲
>>> c = c + b ↲
```

¿Qué se mostrará por pantalla si imprimimos *a*, *b* y *c* al final?