6.5.3. Más sobre el paso de parámetros

Hemos visto que el paso de parámetros comporta que cada parámetro apunte a un lugar de la memoria y que éste puede estar ya apuntado por una variable o parámetro perteneciente al ámbito desde el que se produce la llamada. ¿Qué ocurre si el parámetro es modificado dentro de la función? ¿Se modificará igualmente la variable o parámetro del ámbito desde el que se produce la llamada? Depende. Estudiemos unos cuantos ejemplos.

Para empezar, uno bastante sencillo:

parametros_5.py — parametros.py

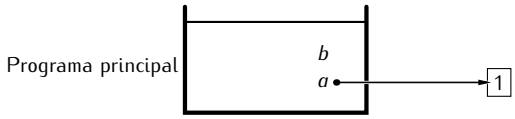
```
1 def incrementa(p):
2     p = p + 1
3     return p
4
5 a = 1
6 b = incrementa(a)
7
8 print 'a:', a
9 print 'b:', b
```

Veamos qué sale por pantalla al ejecutarlo:

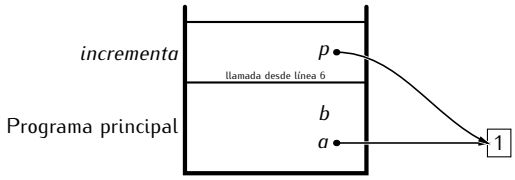
```
a: 1
b: 2
```

Puede que esperaras que tanto *a* como *b* tuvieran el mismo valor al final: a fin de cuentas la llamada a *incrementa* en la línea 6 hizo que el parámetro *p* apuntara al mismo lugar que *a* y esa función incrementa el valor de *p* en una unidad (línea 2). ¿No debería, pues, haberse modificado el valor de *a*? No.

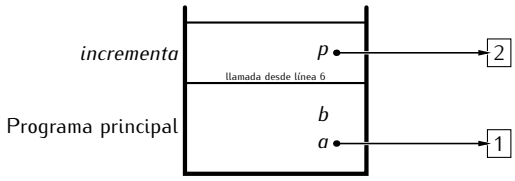
Veamos qué ocurre paso a paso. Inicialmente tenemos en la pila la reserva de memoria para las variables *a* y *b*. Tras ejecutar la línea 5, *a* tiene por valor el entero 1:



Cuando llamamos a *incrementa* el parámetro *p* recibe *una referencia al valor apuntado por a*. Así pues, tanto *a* como *p* apuntan al mismo lugar y valen 1:



El resultado de ejecutar la línea 2 ¡hace que *p* apunte a *una nueva zona de memoria* en la que se guarda el valor 2!

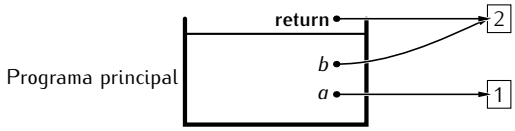


¿Por qué? Recuerda cómo procede Python ante una asignación:

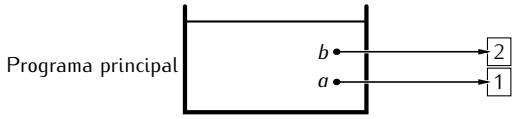
- en primer lugar se evalúa la expresión a mano derecha del igual,
- y a continuación se hace que la parte izquierda del igual apunte al resultado.

La evaluación de una expresión proporciona una referencia a la zona de memoria que alberga el resultado. Así pues, la asignación tiene un efecto sobre la referencia de *p*, no sobre el contenido de la zona de memoria apuntada por *p*. Cuando Python ha evaluado la parte derecha de la asignación de la línea 2, ha sumado al valor 1 apuntado por *p* el valor 1 que aparece explícitamente. El resultado es 2, así que Python ha reservado una nueva celda de memoria con dicho valor. Finalmente, se ha asignado a *p* el resultado de la expresión, es decir, se ha hecho que *p* apunte a la celda de memoria con el resultado.

Sigamos con la ejecución de la llamada a la función. Al finalizar ésta, la referencia de *p* se devuelve y, en la línea 6, se asigna a *b*.



Resultado: *b* vale lo que valía *p* al final de la llamada y *a* no ve modificado su valor:



EJERCICIOS

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

parametros_6.py — parametros.py

```
1 def incrementa(a):
2     a = a + 1
3     return a
4
```

```
5 a = 1
6 b = incrementa(a)
7
8 print 'a:', a
9 print 'b:', b
```

Hazte un dibujo del estado de la pila de llamadas paso a paso para entender bien qué está pasando al ejecutar cada sentencia.

Y ahora, la sorpresa:

paso_de_listas.py

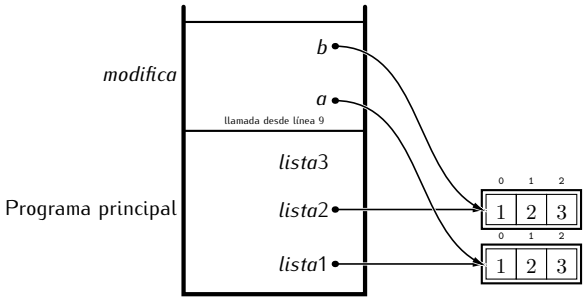
```
1  def modifica(a, b):
2      a.append(4)
3      b = b + [4]
4      return b
5
6  lista1 = [1, 2, 3]
7  lista2 = [1, 2, 3]
8
9  lista3 = modifica(lista1, lista2)
10
11 print lista1
12 print lista2
13 print lista3
```

Ejecutemos el programa:

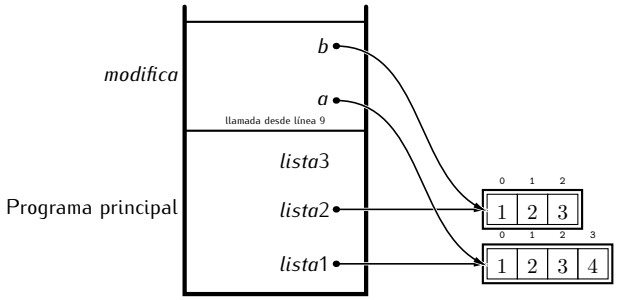
```
[1, 2, 3, 4]
[1, 2, 3]
[1, 2, 3, 4]
```

¿Qué ha ocurrido? La lista que hemos proporcionado como primer argumento se ha modificado al ejecutarse la función y la que sirvió de segundo argumento no.

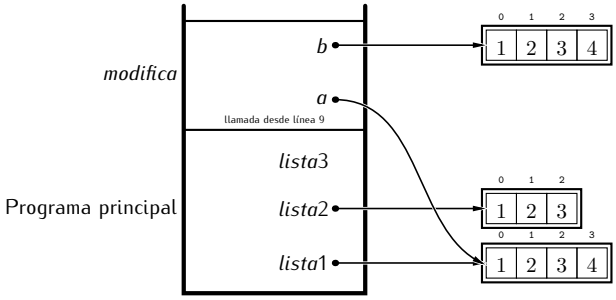
Ya deberías tener suficientes datos para averiguar qué ha ocurrido. No obstante, nos detendremos brevemente a explicarlo. Veamos en qué estado está la memoria en el momento en el que se produce el paso de parámetros en la llamada a *modifica*:



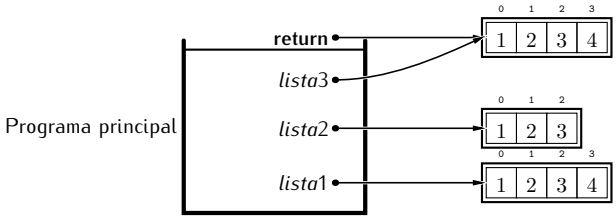
¿Qué ocurre cuando se ejecuta la línea 2? Que la lista apuntada por *a* crece por el final (con *append*) con un nuevo elemento de valor 4:



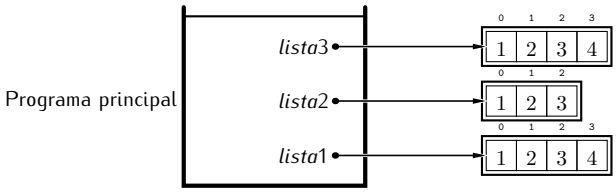
Como esa lista está apuntada tanto por el parámetro *a* como por la variable global *lista1*, ambos «sufren» el cambio y ven modificado su valor. Pasemos ahora a la línea 3: una asignación. Como siempre, Python empieza por evaluar la parte derecha de la asignación, donde se indica que se debe crear *una nueva lista* con capacidad para cuatro elementos (los valores 1, 2 y 3 que provienen de *b* y el valor 4 que aporta la lista `[4]`). Una vez creada la nueva lista, se procede a que la variable de la parte izquierda apunte a ella:



Cuando finaliza la ejecución de *modifica*, *lista3* pasa a apuntar a la lista devuelta por la función, es decir, a la lista que hasta ahora apuntaba *b*:



Y aquí tenemos el resultado final:



Recuerda, pues, que:

- La asignación puede comportar un cambio del lugar de memoria al que apunta una variable. *Si un parámetro modifica su valor mediante una asignación, (probablemente) obtendrá una nueva zona de memoria y perderá toda relación con el argumento del que tomó valor al efectuar el paso de parámetros.*
- *Operaciones como append*, **del** *o la asignación a elementos indexados de listas modifican a la propia lista, por lo que los cambios afectan tanto al parámetro como al argumento.*

Con las cadenas ocurre algo similar a lo estudiado con las listas, solo que las cadenas son inmutables y no pueden sufrir cambio alguno mediante operaciones como *append*, **del** o asignación directa a elementos de la cadena. De hecho, ninguna de esas operaciones es válida sobre una cadena.

EJERCICIOS

▶ 339 ¿Qué mostrará por pantalla el siguiente programa al ejecutarse?

ejercicio_parametros_4.py — ejercicio_parametros.py

```
1 def modifica(a, b):
2     for elemento in b:
```

```
3       a.append(elemento)
4     b = b + [4]
5     a[-1] = 100
6     del b[0]
7     return b[:]
8
9   lista1 = [1, 2, 3]
10  lista2 = [1, 2, 3]
11
12  lista3 = modifica(lista1, lista2)
13
14  print lista1
15  print lista2
16  print lista3
```

▶ 340 ¿Qué muestra por pantalla este programa al ser ejecutado?

ejercicio_parametros_5.py — ejercicio_parametros.py

```
1  def modifica_parametros(x, y):
2      x = 1
3      y[0] = 1
4
5  a = 0
6  b = [0, 1, 2]
7  modifica_parametros(a, b)
8
9  print a
10 print b
```

▶ 341 ¿Qué muestra por pantalla este programa al ser ejecutado?

ejercicio_parametros_6.py — ejercicio_parametros.py

```
1  def modifica_parametros(x, y):
2      x = 1
3      y.append(3)
4      y = y + [4]
5      y[0] = 10
6
7  a = 0
8  b = [0, 1, 2]
9  modifica_parametros(a, b)
10 print a
11 print b
```

▶ 342 Utiliza las funciones desarrolladas en el ejercicio 307 y diseña nuevas funciones para construir un programa que presente el siguiente menú y permita ejecutar las acciones correspondientes a cada opción:

```
1) Añadir estudiante y calificación
2) Mostrar lista de estudiantes con sus calificaciones
3) Calcular la media de las calificaciones
4) Calcular el número de aprobados
5) Mostrar los estudiantes con mejor calificación
6) Mostrar los estudiantes con calificación superior a la media
7) Consultar la nota de un estudiante determinado
8) FINALIZAR EJECUCIÓN DEL PROGRAMA
```

Ahora que sabemos que dentro de una función podemos modificar listas vamos a diseñar una función que invierta una lista. ¡Ojo!: no una función que, dada una lista, *devuelva* otra que sea la inversa de la primera, sino un procedimiento (recuerda: una función que no devuelve nada) que, dada una lista, *la modifique* invirtiéndola.

El aspecto de una primera versión podría ser éste:

inversion_4.py — inversion.py

```
1 def invierte(lista):
2     for i in range(len(lista)):
3         intercambiar los elementos lista[i] y lista[len(lista)-1-i]
```

Intercambiaremos los dos elementos usando una variable auxiliar:

inversion_5.py — inversion.py

```
1 def invierte(lista):
2     for i in range(len(lista)):
3         c = lista[i]
4         lista[i] = lista[len(lista)-1-i]
5         lista[len(lista)-1-i] = c
6
7 a = [1, 2, 3, 4]
8 invierte(a)
9 print a
```

Ejecutemos el programa:

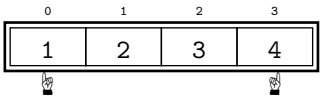
```
[1, 2, 3, 4]
```

No funciona. Parece que no la haya modificado. En realidad sí que lo ha hecho, pero mal. Estudiemos paso a paso qué ha ocurrido:

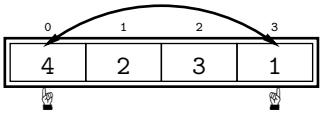
1. Al llamar a la función, el parámetro *lista* «apunta» (hace referencia) a la misma zona de memoria que la variable *a*.

2. El bucle que empieza en la línea 2 va de 0 a 3 (pues la longitud de *lista* es 4). La variable local *i* tomará los valores 0, 1, 2 y 3.

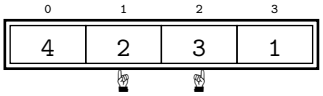
   a) Cuando *i* vale 0, el método considera los elementos *lista*[0] y *lista*[3]:

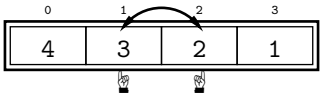
   La variable local *c* toma el valor 1 (que es el contenido de *lista*[0]), a continuación *lista*[0] toma el valor de *lista*[3] y, finalmente, *lista*[3] toma el valor de *c*. El resultado es que se intercambian los elementos *lista*[0] y *lista*[3]:

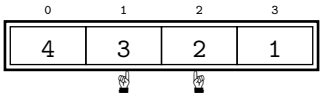
   b) Ahora *i* vale 1, así que se consideran los elementos *lista*[1] y *lista*[2]:

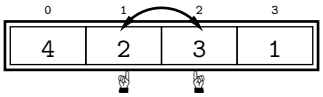
   Los dos elementos se intercambian y la lista queda así:



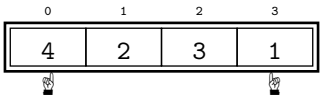
c) Ahora *i* vale 2, así que se consideran los elementos *lista*[2] y *lista*[1]:



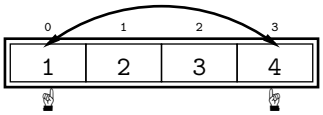
Tras el intercambio, la lista pasa a ser:



d) Y, finalmente, *i* vale 3.



Se intercambian los valores de las celdas *lista*[3] y *lista*[0]:



Fíjate en que al final de la segunda iteración del bucle la lista estaba correctamente invertida. Lo que ha ocurrido es que hemos seguido iterando y ¡hemos vuelto a invertir una lista que ya estaba invertida, dejándola como estaba al principio! Ya está claro cómo actuar: iterando la mitad de las veces. Vamos allá:

inversion.py

```
def invierte(lista):
    for i in range(len(lista)/2):
        c = lista[i]
        lista[i] = lista[len(lista)-1-i]
        lista[len(lista)-1-i] = c

a = [1, 2, 3, 4]
invierte(a)
print a
```

Ahora sí. Si ejecutamos el programa obtenemos:

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

EJERCICIOS

▶ 343 ¿Qué ocurre con el elemento central de la lista cuando la lista tiene un número impar de elementos? ¿Nuestra función invierte correctamente la lista?

▶ 344 Un aprendiz sugiere esta otra solución. ¿Funciona?

inversion_6.py

```
def invierte(lista):
    for i in range(len(lista)/2):
        c = lista[i]
        lista[i] = lista[-i-1]
        lista[-i-1] = c
```

▶ 345 ¿Qué muestra por pantalla este programa al ser ejecutado?

abslista_2.py — abslista.py

```
def abs_lista(lista):
    for i in range(len(lista)):
        lista[i] = abs(lista[i])

milista = [1, -1, 2, -3, -2, 0]
abs_lista(milista)
print milista
```

▶ 346 ¿Qué mostrará por pantalla el siguiente programa al ejecutarse?

intercambio_2.py — intercambio.py

```
def intento_de_intercambio(a, b):
    aux = a
    a = b
    b = aux

lista1 = [1, 2]
lista2 = [3, 4]

intento_de_intercambio(lista1, lista2)

print lista1
print lista2
```

▶ 347 Diseña un procedimiento que, dada una lista de números, la modifique para que sólo sobrevivan a la llamada aquellos números que son perfectos.

▶ 348 Diseña una función *duplica* que reciba una lista de números y la modifique duplicando el valor de cada uno de sus elementos. (Ejemplo: la lista `[1, 2, 3]` se convertirá en `[2, 4, 6]`.)

▶ 349 Diseña una función *duplica_copia* que reciba una lista de números y devuelva *otra* lista en la que cada elemento sea el doble del que tiene el mismo índice en la lista original. La lista original *no debe sufrir ninguna modificación* tras la llamada a *duplica_copia*.

▶ 350 Diseña una función que reciba una lista y devuelva otra lista cuyo contenido sea el resultado de concatenar la lista original consigo misma. La lista original no debe modificarse.

▶ 351 Diseña una función que reciba una lista y devuelva otra lista cuyo contenido sea la lista original, pero con sus componentes en orden inverso. La lista original no debe modificarse.

▶ 352 Diseña una función que reciba una lista y devuelva una copia de la lista concatenada con una inversión de sí misma. Puedes utilizar, si lo consideras conveniente, funciones que has desarrollado en ejercicios anteriores.

▶ 353 Diseña una función que reciba una lista y devuelva una lista cuyo contenido sea la lista original concatenada con una versión invertida de ella misma. La lista original no debe modificarse.

▶ 354 Diseña una función que reciba una lista y devuelva una copia de la lista con sus elementos ordenados de menor a mayor. La lista original no debe modificarse.

▶ 355 Diseña un procedimiento que reciba una lista y ordene sus elementos de menor a mayor.

▶ 356 Diseña una función que reciba una matriz y, si es cuadrada (es decir, tiene igual número de filas que de columnas), devuelva la suma de todos los componentes dispuestos en la diagonal principal (es decir, todos los elementos de la forma $A_{i,i}$). Si la matriz no es cuadrada, la función devolverá *None*.

▶ 357 Guardamos en una matriz de $m \times n$ elementos la calificación obtenida por m estudiantes (a los que conocemos por su número de lista) en la evaluación de n ejercicios entregados semanalmente (cuando un ejercicio no se ha entregado, la calificación es -1).

Diseña funciones y procedimientos que efectúen los siguiente cálculos:

- Dado el número de un alumno, devolver el número de ejercicios entregados.
- Dado el número de un alumno, devolver la media sobre los ejercicios entregados.
- Dado el número de un alumno, devolver la media sobre los ejercicios entregados si los entregó todos; en caso contrario, la media es 0.
- Devolver el número de todos los alumnos que han entregado todos los ejercicios y tienen una media superior a 3.5 puntos.
- Dado el número de un ejercicio, devolver la nota media obtenida por los estudiantes que lo presentaron.
- Dado el número de un ejercicio, devolver la nota más alta obtenida.
- Dado el número de un ejercicio, devolver la nota más baja obtenida.
- Dado el número de un ejercicio, devolver el número de estudiantes que lo han presentado.
- Devolver el número de abandonos en función de la semana. Consideramos que un alumno abandonó en la semana x si no ha entregado ningún ejercicio desde entonces. Este procedimiento mostrará en pantalla el número de abandonos para cada semana (si un alumno no ha entregado nunca ningún ejercicio, abandonó en la «semana cero»).