


Status	Finished
Started	Saturday, 23 November 2024, 5:06 PM
Completed	Saturday, 23 November 2024, 5:13 PM
Duration	7 mins 21 secs
Marks	2.50/3.00
Grade	8.33 out of 10.00 (83.33%)

Information

Alias


Una lista puede ser [referenciada](#) por más de una [variable](#):

```
a = [1, 2, 3]
b = a
```




Las [variables](#) a y b [referencian](#) la [misma](#) lista. Decimos que ambas [variables](#) son [alias](#) del mismo [objeto](#) (una lista, en este caso). Esto significa que si se modifica el [objeto referenciado](#) por una de las [variables](#) (la lista en cuestión), el [objeto referenciado](#) por la otra (que es el mismo) lo veríamos asimismo modificado. En el siguiente ejemplo, tras asignar `b[1]=0`, veríamos que el elemento `a[1]` pasaría a valer 0, y no 2, como originalmente.

```
b[1] = 0
```



Si se necesita tener una copia independiente, se puede usar la [función](#) `copy()`, del módulo `copy`, que hay que importar. Como resultados tendremos dos listas que, aunque de momento son [iguales](#), no son la misma lista (no son el [mismo objeto](#)). Por tanto, ahora la [asignación](#) `b[1]=0` no tendría efecto sobre `a[1]`, que seguiría valiendo 2.

```
a = [1, 2, 3]
b = copy(a)
```



Question 1

Complete

Mark 0.50 out of 1.00

Marque las afirmaciones correctas.

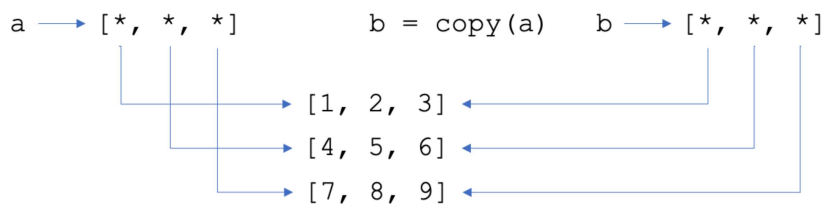
Select one or more:

- ☒ Las listas son una estructura de datos mutable
- ☒ Dos listas diferentes pueden ser [referenciadas](#) directamente por la misma [variable](#) al mismo tiempo
- ☐ Una lista puede ser [referenciada](#) por 10 [variables](#) diferentes

Information

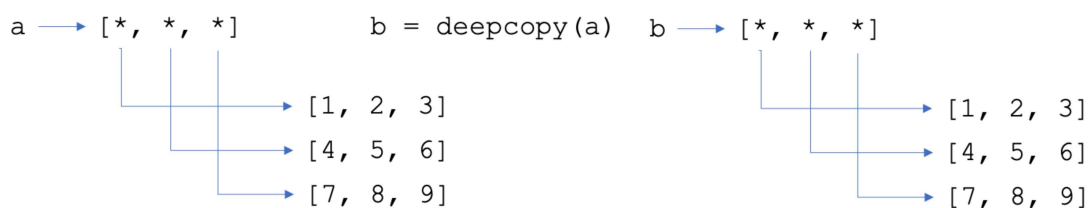
Shallow copy y deep copy

La [función](#) `copy()` del módulo `copy` realiza una copia "superficial" o "somera". Esto significa que si los elementos de la lista, en vez de [referenciar valores](#) de un tipo simple, como números [enteros](#), son a su vez [referencias](#) a [objetos](#) (p.e., una lista de listas), los elementos de 2º nivel (y sucesivos niveles, si los hubiera) quedan compartidos, ya que sólo se copian las [referencias](#) de 1º nivel:

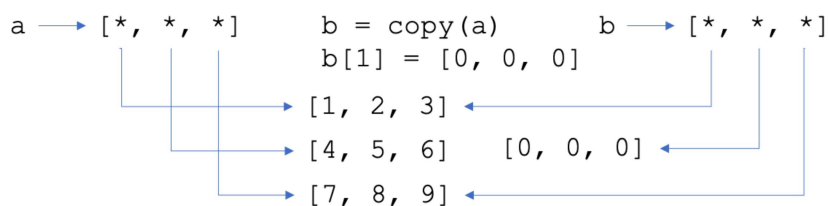


Si asignamos `a[0][1]=5`, el [valor](#) al que accederíamos a continuación mediante `b[0][1]` lo veríamos también modificado, ya que `a[0]` y `b[0]` [referencian](#) el mismo [objeto](#) lista.

Si se quiere hacer una copia "profunda", hay que usar la [función](#) `deepcopy()` en vez de la `copy()`:



De todas formas, la compartición de elementos de segundo nivel y no del primero, permite modificar elementos del primer nivel sin que se modifiquen los correspondientes del segundo de la otra [variable](#):



Information

Paso de listas como **parámetros**

Cuando tenemos que procesar una lista, frecuentemente la pasamos como [parámetro](#) a una [función](#). En estos casos el [parámetro](#) formal de la [función](#) [referenciará](#) a la [misma](#) lista que pasamos en la llamada.

La [función](#) del siguiente ejemplo devuelve el [valor](#) máximo almacenado en una lista, de hecho, en cualquier secuencia iterable (como por ejemplo una tupla o una [lista](#)). Podemos además ver que para esta [función](#) ejemplo los elementos de la secuencia iterable deben ser comparables entre sí, ya que no funcionaría si mezclamos datos de p.e. tipos int y str:

```
def maximum(a):
    m = a[0]
    for i in range(1, len(a)):
        if a[i] > m:
            m = a[i]
    return m

data = [1, 3, 8, 5, 6]
print(maximum(data))      # 8
print(data)               # [1, 3, 8, 5, 6]
print(maximum('Python'))  # y
```

Obsérvese que en el último caso hemos usado *maximum* con una *str*, secuencia iterable compuesta de caracteres.

En el ejemplo anterior no hemos modificado la lista recibida por [parámetro](#). Por el contrario, en el ejemplo siguiente la [función](#) intercambia dos elementos de una lista (se supone que tanto *i* como *j* son [índices](#) válidos para la lista *a*):

```
def exchange(a, i, j):
    a[i], a[j] = a[j], a[i]

values = [2, 6, 8, 5, 6, 9]
exchange(values, 1, 3)
print(values)          # [2, 5, 8, 6, 6, 9]
```

En ambos ejemplos, el [parámetro](#) formal, *a*, es una [referencia](#) al [parámetro](#) real que se pasa en la llamada, que en el segundo ejemplo es *values*, por lo que, al modificar la lista usando *a* en la [función](#), se está modificando la [misma](#) lista [referenciada](#) por *values* desde [fuera](#) de la [función](#). En este caso a *exchange*, a diferencia de *maximum*, solo se le puede pasar una secuencia mutable.

Question 2

Complete

Mark 1.00 out of 1.00

Queremos una [función](#) para borrar todos los elementos de una lista. ¿Cuál de las dos opciones proporcionadas en el siguiente código es correcta?

```
def clear1(list):
    list = []

def clear2(list):
    list.clear()

# Compare los resultados de las siguiente llamadas:
a = list('Python')
print(a)          # ['P', 'y', 't', 'h', 'o', 'n']
clear1(a)
print(a)

a = list('Python')
print(a)          # ['P', 'y', 't', 'h', 'o', 'n']
clear2(a)
print(a)
```

Select one:

- ☒ clear2()
- ☐ clear1()

Information

Listas como resultado de funciones

Además de poder pasar [referencias](#) a listas como [parámetros](#), las [referencias](#) a listas también pueden [devolverse](#) como resultado de una [función](#):

```
def create_list(n, value = 0):
    return [value] * n

print(create_list(10))      # [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
print(create_list(5, -1))   # [-1, -1, -1, -1, -1]
print(create_list(8, None)) # [None, None, None, None, None, None, None, None]
print(create_list(5, 'Hi')) # ['Hi', 'Hi', 'Hi', 'Hi', 'Hi']
```

En tanto en cuanto no especifiquemos el tipo del resultado esperado y no usemos una herramienta para tenerlo en cuenta, la [función](#) `create_list()` es una plantilla general para crear listas de elementos con un [valor](#) determinado.

Para [especificar](#) que se requiere una lista como [parámetro](#) o se espera como resultado podemos usar, igual que para los tipos básicos, el nombre de tipo `list`. Si quisiéramos [especificar](#) además de qué tipo deben ser, a su vez, los elementos de la lista, podemos importar del módulo `typing` el nombre `List`, como en el siguiente ejemplo:

```
from typing import List

def create_list(n: int, value: int = 0) -> List[int]:
    return [value] * n
```

Para las tuplas (tipos `tuple` y `typing.Tuple`) ocurre igual.

Information

Funciones modificadoras y no modificadoras

Cuando escribimos funciones que manejan listas, debemos tener presente si queremos que la lista original se modifique o no.

```
def add1(list, element):  
    return list + [element]  
  
def add2(list, element):  
    list.append(element)  
    return list  
  
data = [1, 3, 8, 5, 6]  
print(add1(data, 1000))      # [1, 3, 8, 5, 6, 1000]  
print(data)                  # [1, 3, 8, 5, 6]  
print(add2(data, 1000))      # [1, 3, 8, 5, 6, 1000]  
print(data)                  # [1, 3, 8, 5, 6, 1000]
```

En el ejemplo anterior, la [función add1\(\)](#) devuelve una [referencia](#) a una nueva lista, sin modificar la original, mientras que la [función add2\(\)](#) añade un elemento a la lista original y devuelve la [referencia](#) a la lista original ya modificada. Decimos que la [función add1\(\)](#) es no modificadora y la [función add2\(\)](#) es modificadora.

Question 3

Complete

Mark 1.00 out of 1.00

Necesitamos invertir en una lista el orden de sus elementos. ¿Cuál de las siguientes operaciones es *modificadora*?

```
def f1(a):  
    return a[::-1]  
  
def f2(a):  
    a[:] = a[::-1]  
    return a  
  
def f3(a):  
    b = []  
    for x in a:  
        b = [x] + b  
    return b
```

Select one:

- ☒ f2()
☐ f1()
☐ f3()