Tabla de contenidos

2. El Modelo Mental Correcto

Si vienes de Java o C, probablemente piensas en las variables como cajas. Declaras `int x = 10` y la variable `x` parece una caja que contiene el valor `10`, con un cartel que dice "esto es un entero". Si luego escribes `x = "hola"`, el compilador te dice que no puedes meter un `String` en una caja de enteros: el tipo pertenece a la caja.

Python no funciona así. En Python, las variables no son cajas: son **etiquetas**. El tipo no está en la variable, está en el objeto.

Este es el cambio de perspectiva más importante del capítulo. El tipado dinámico, la mutabilidad, los efectos secundarios y la diferencia entre `is` y `==` se entienden mejor a partir de este modelo.

2.1. De cajas a etiquetas

Mira estos dos fragmentos. Son equivalentes en lo que hacen, pero radicalmente distintos en lo que significan.

Java:

```
int myBox = 10;
myBox = "hola"; // ERROR: incompatible types
```

Python:

```
my_label = 10
my_label = "hola"  # Correcto: la etiqueta se pega al nuevo objeto
```

En Java, el compilador rechaza la segunda línea porque `myBox` se declaró como `int` y no puede contener otra cosa. Para este tipo primitivo, la metáfora de la caja resulta útil.

Java también maneja referencias: `List<Integer> b = a` no copia la lista y `b.add(1)` se observa mediante `a`. El aliasing no es exclusivo de Python. La diferencia central es que un nombre de Python puede enlazarse después con un objeto de otro tipo.

En Python, `my_label` nunca fue "de tipo int". Fue una etiqueta que apuntaba a un objeto `10` (que resulta ser de tipo `int`). Luego la despegamos y la pegamos a un objeto `"hola"` (que resulta ser de tipo `str`). La etiqueta no tiene tipo — **el objeto lo tiene**.

> **Información**
>
> **Regla de oro:** en Python, todo es un objeto: números, cadenas, listas, funciones, clases. Las variables solo son nombres que apuntan a esos objetos. Cuando asignas, no copias el objeto: pegas la etiqueta.

2.1.1. El diagrama mental

Dos etiquetas pueden apuntar a objetos distintos, o al mismo objeto:

```
x = 42      # etiqueta x → objeto int 42
y = "Ana"   # etiqueta y → objeto str "Ana"  (cada etiqueta apunta a
 ↪  un objeto distinto)

a = [1, 2, 3]
b = a       # dos etiquetas, un objeto: b = a no copia la lista, solo
 ↪  añade otra etiqueta
```

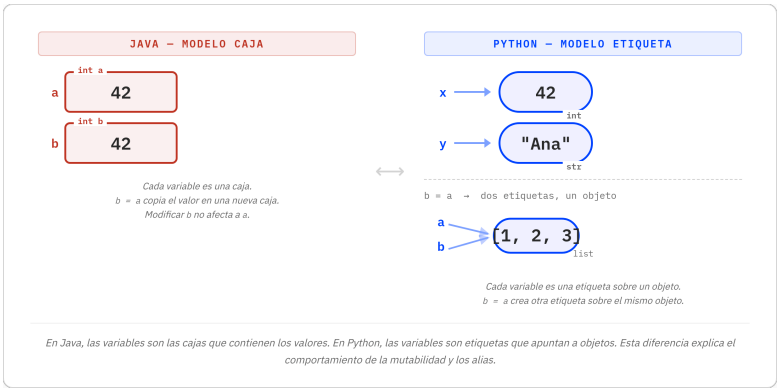

Figura 2.1.: Modelo de etiquetas de Python.

`x` e `y` apuntan a objetos distintos; `a` y `b` apuntan al mismo. Si modificas la lista a través de `b`, el cambio también es visible desde `a` — no hay dos listas, solo dos etiquetas sobre una.

Python puede liberar un objeto cuando ya no quedan referencias alcanzables a él. El mecanismo concreto depende de la implementación; para razonar sobre el programa basta seguir qué nombres y contenedores lo referencian.

2.1.2. Las dos herramientas de inspección

El modelo de etiquetas no es solo teoría: Python te permite verificarlo en tiempo de ejecución. Cuando dudas de qué tipo tiene un objeto o de si dos variables apuntan al mismo lugar en memoria, no necesitas adivinar — puedes preguntárselo directamente al intérprete con dos funciones built-in:

```
a = 42
print(type(a))   # el tipo está en el objeto, no en la variable
```

```
<class 'int'>
125368008361488
```

- `type()` te dice qué tipo tiene el objeto al que apunta la etiqueta.
- `id()` devuelve un entero que identifica al objeto mientras este existe. En CPython suele coincidir con su dirección de memoria, pero Python no garantiza que sea una dirección.

Tabla 2.1.: Funciones de inspección de Python. `type()` y `id()` revelan el modelo de etiquetas en acción.

Función	Te dice	Analogía Java
`type(x)`	Tipo del objeto	`x.getClass()`
`id(x)`	Identificador de identidad durante la vida del objeto	`System.identityHashCode(x)`, aunque este es un hash y puede colisionar

Error Común

Error común del programador Java: intentar adivinar el tipo de una variable mirando el código en lugar de usar `type()`. En Python no declaras tipos, así que la única forma de estar seguro es preguntarle al objeto. Usa `type()` cuando dudes.

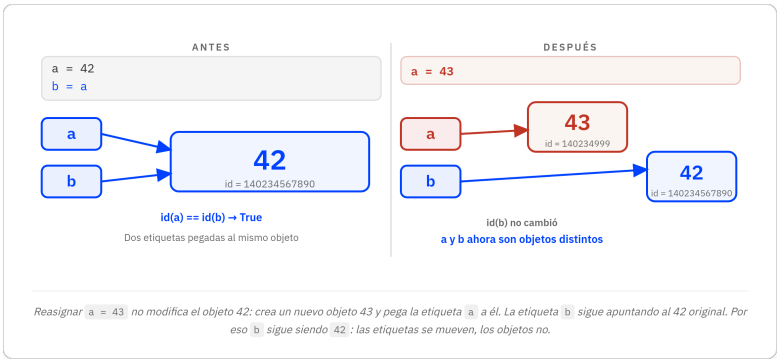

Figura 2.2.: Modelo de memoria.

Manos a la Obra

Manos a la obra — Ejercicio 1: Las etiquetas en acción
Copia y ejecuta este código en una celda:

```
a = 42
b = a
print("id(a):", id(a))
print("id(b):", id(b))
print("id(a) == id(b):", id(a) == id(b))
```

Antes de ejecutar la siguiente celda, predice qué ocurrirá:

```
a = 43
print("id(a):", id(a))
print("id(b):", id(b))
print("b sigue siendo:", b)
```

Responde en una celda Markdown:

1. ¿Por qué `id(a) == id(b)` es `True` antes de cambiar `a`?
2. ¿Por qué `b` sigue siendo `42` después de cambiar `a` a `43`?

Criterio de éxito: tus respuestas mencionan explícitamente el modelo de etiquetas: “a y b eran dos etiquetas pegadas al mismo objeto 42; al asignar a = 43, la etiqueta a se pegó a otro objeto, pero b sigue apuntando a 42”.

2.2. Identidad vs. igualdad

Si las variables son etiquetas, dos etiquetas pueden apuntar al mismo objeto (misma identidad) o a objetos distintos con el mismo contenido (misma

igualdad). La distinción es tan importante que Python tiene dos operadores distintos.

- `==` compara el **contenido** de los objetos. Pregunta: "¿valen lo mismo?"
- `is` compara la **identidad** de los objetos. Pregunta: "¿son el mismo objeto en memoria?"

2.2.1. Demostración con listas

Dos listas con el mismo contenido son iguales (`==`) pero no idénticas (`is`) — son objetos distintos en memoria. Una tercera variable que apunta a la misma lista sí es idéntica:

```
list_a = [1, 2, 3]
list_b = [1, 2, 3]  # mismo contenido, objeto distinto
list_c = list_a     # dos nombres, un mismo objeto

print(list_a == list_b)  # True — mismo contenido
print(list_a is list_b)  # False — objetos distintos, distinto id()
print(list_a is list_c)  # True — mismo objeto, mismo id()
```

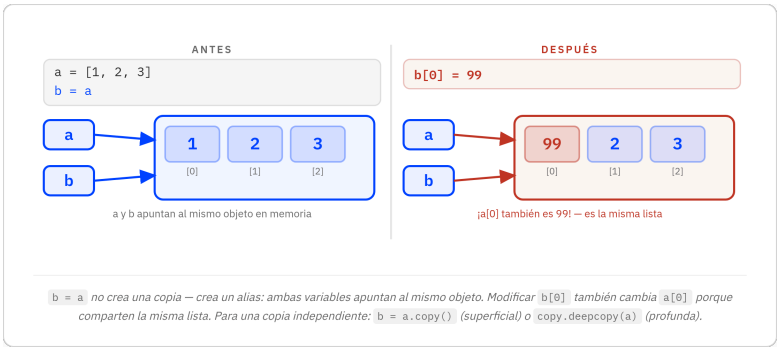

Figura 2.3.: Alias y mutación.

list_a y list_c apuntan al mismo objeto (X). list_b apunta a otro objeto (Y) que tiene el mismo contenido. == dice que son iguales; `is` dice que son distintos.

2.2.2. El caso de uso principal: `None`

En Python profesional, el uso más habitual de `is` es comparar con `None`:

```
result = some_function()
if result is None:   # correcto (estándar)
```

```
    print("No hay resultado")
```

No uses `result == None`: además de ser semánticamente incorrecto, puede invocar una comparación sobrecargada del objeto. Los linters también lo señalan como mala práctica.

Información

`None` es un objeto **singleton**: solo hay una instancia durante la ejecución. Por eso `is None` expresa exactamente la pregunta correcta: «¿es este objeto `None`?».
Usa `==` para comparar valores, incluidos números y cadenas. Que algunos literales compartan identidad por cachés o *interning* es un detalle de implementación y no justifica usar `is` con ellos.

Error Común

Error común del programador Java: para *objetos*, la correspondencia es aproximadamente inversa: `==` en Python ≈ `.equals()` en Java, y `is` en Python ≈ `==` en Java para referencias. Para tipos primitivos de Java, `==` compara valores, igual que `==` en Python para `int` o `str`.

Manos a la Obra

Manos a la obra — Ejercicio 2: Identidad vs. igualdad en la práctica
Completa esta tabla de predicciones ANTES de ejecutar el código:

```
x = [1, 2, 3]
y = x
z = x.copy()
w = [1, 2, 3]
```

Tabla 2.2.: Tabla de predicciones para identidad vs. igualdad.

Comparación	Tu predicción	Resultado al ejecutar
`x == y`		
`x is y`		
`x == z`		
`x is z`		
`x == w`		
`x is w`		

Después de ejecutar, responde: ¿por qué `x is z` es `False` aun cuando `x == z` es `True`? Explica con el diagrama de memoria.
Criterio de éxito: aciertas al menos 5 de 6 predicciones. La explicación menciona que `.copy()` crea un nuevo objeto con contenido idéntico pero distinta identidad.

2.3. Mutabilidad e inmutabilidad

Ahora que sabes que las variables son etiquetas y que dos etiquetas pueden apuntar al mismo objeto, la pregunta siguiente es: ¿qué pasa si **modificas** el objeto al que apuntan dos etiquetas?

La respuesta depende del tipo del objeto. Este modelo se reutiliza al razonar sobre las colecciones mutables e inmutables de Capítulo 4 y explica por qué las funciones pueden tratarse como objetos en Capítulo 5.

2.3.1. Tipos mutables vs. inmutables

Tabla 2.3.: Clasificación de tipos de Python por mutabilidad.

Si es ...	Característica	Ejemplos
Inmutable	No se puede modificar. Cualquier operación que parezca modificarlo crea un nuevo objeto. Una `tuple` puede contener objetos mutables.	`int`, `float`, `complex`, `str`, `tuple`
Mutable	Se puede modificar en el sitio. El objeto cambia pero sigue siendo el mismo en memoria.	`list`, `dict`, `set`

2.3.2. Demostración con inmutables

Observa cómo se comporta un entero (inmutable). ¿Qué crees que ocurrirá tras `x = x + 1`?

```
x = 10
y = x
print(id(x) == id(y))  # True — dos nombres, un mismo objeto

x = x + 1
```

```
print(x)                # 11
print(y)                # 10 — y no cambió
print(id(x) == id(y))   # False — x ahora apunta a otro objeto
```

Cuando haces `x = x + 1`, Python no modifica el objeto `10`. Crea un nuevo objeto `11` y pega la etiqueta `x` a él. La etiqueta `y` sigue apuntando a `10`.

2.3.3. Demostración con mutables

Ahora observa cómo se comporta una lista (mutable):

```
list_a = [1, 2, 3]
list_c = list_a          # NO es una copia
print(id(list_a) == id(list_c))  # True — misma lista

list_c.append(99)
print(list_c)            # [1, 2, 3, 99]
print(list_a)            # [1, 2, 3, 99] — ¡list_a también cambió!
print(id(list_a) == id(list_c))  # True — sigue siendo el mismo objeto
```

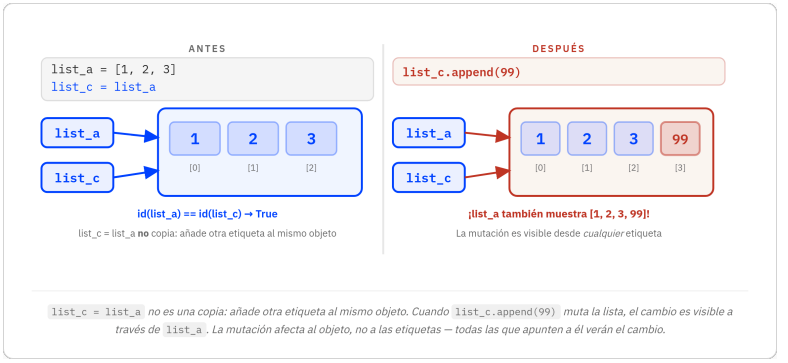

Figura 2.4.: Mutación compartida.

Esto es el bug clásico de mutabilidad. Cuando asignas `list_c = list_a`, no estás haciendo una copia. Estás pegando la etiqueta `list_c` al mismo objeto al que apunta `list_a`. Al modificar el objeto a través de una etiqueta, todos los cambios son visibles a través de cualquier otra etiqueta que apunte al mismo objeto.

> **Error Común**
>
> **Error común del programador Java (y de todos):** pensar que `list_c = list_a` hace una copia. No la hace. Especialmente peligroso cuando pasas una lista como argumento a una función: la función puede modificar la lista original

sin que el llamador lo espere. Es una de las fuentes más frecuentes de bugs silenciosos en Python.

2.3.4. La solución: copia explícita

Si quieres un contenedor independiente, debes pedir una copia explícita. `.copy()` crea una copia superficial: basta para esta lista plana, pero no aísla objetos mutables anidados:

```
list_a = [1, 2, 3]
list_c = list_a.copy()  # crea un nuevo objeto lista con el mismo
↪  contenido
list_c.append(99)
print(list_a)   # [1, 2, 3] — no cambió
print(list_c)   # [1, 2, 3, 99] — solo cambió la copia
```

O, equivalentemente:

```
list_c = list(list_a)  # misma funcionalidad que .copy()
```

Pro-Tip

Consejo profesional: si recibes una lista o diccionario y no quieres modificar su contenedor original, puedes empezar con `datos = datos_original.copy()`. Comprueba antes su profundidad: los mutables anidados siguen compartidos. Copia solo las ramas que vayas a modificar o usa `copy.deepcopy()` cuando necesites aislar toda la estructura copiable.

Manos a la Obra

Manos a la obra — Ejercicio 3: Depura el efecto secundario
El contrato exige que esta función devuelva una lista nueva sin modificar la recibida. Sin ejecutar, identifica por qué lo incumple:

```
def duplicate_values(numbers):
    numbers[:] = [n * 2 for n in numbers]
    return numbers

originals = [10, 20, 30]
result = duplicate_values(originals)
print(originals)  # ¿Qué imprime?
print(result)     # ¿Qué imprime?
```

Responde:

1. ¿Qué imprime `print(originals)` después de la llamada?
2. ¿Es ese el comportamiento deseado? ¿Por qué?
3. Escribe una versión corregida que no modifique la lista original:

```
def duplicate_values_fixed(numbers):
    # Tu código aquí
    pass
```

Criterio de éxito: detectas que la función modifica la lista original a través de `numbers[:]` y propones una corrección que crea y devuelve una lista nueva (por ejemplo, `return [n * 2 for n in numbers]`). Reasignar el parámetro con `numbers = [...]` tampoco cambiaría el nombre del llamador: mutar un objeto y reenlazar un nombre local son operaciones distintas.

2.3.5. Copia superficial vs. copia profunda

`copy()` y `list()` hacen **copia superficial** (*shallow copy*): crean un nuevo contenedor pero los elementos internos se comparten. Para listas que contienen objetos mutables (como listas dentro de listas), esto puede ser insuficiente. Compáralo con `copy.deepcopy()`:

```
import copy

nested_list = [[1, 2], [3, 4]]
shallow_copy = nested_list.copy()
deep_copy = copy.deepcopy(nested_list)

nested_list[0].append(99)

print(nested_list)         # [[1, 2, 99], [3, 4]]
print(shallow_copy)        # [[1, 2, 99], [3, 4]] — la sublista se
↪  comparte
print(deep_copy)           # [[1, 2], [3, 4]] — todo es independiente
```

- **Copia superficial:** nuevo contenedor exterior, mismos objetos interiores.
- **Copia profunda:** los objetos mutables copiables se duplican recursivamente; puede reutilizar inmutables y preserva las relaciones internas del grafo.

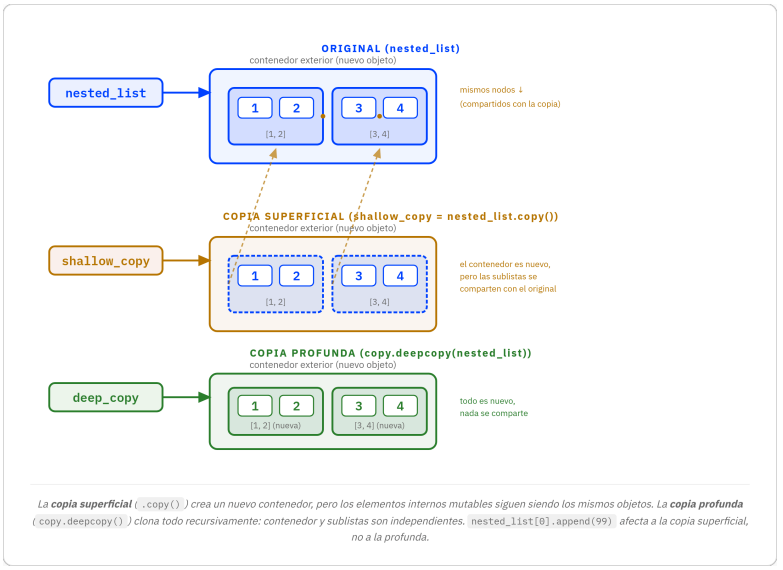

Figura 2.5.: Copia superficial vs. copia profunda.

Tabla 2.4.: Copia superficial vs. copia profunda. Elige según qué partes de la estructura necesites aislar.

Copia	Cómo se hace	Cuándo usarla
Superficial	`.copy()`, `list()`, `dict()`	Listas planas (elementos inmutables)
Profunda	`copy.deepcopy()`	Listas anidadas (elementos mutables dentro)

Manos a la Obra

Manos a la obra — Ejercicio 4: Copia superficial vs. profunda
Ejecuta este código y predice el resultado antes de verlo:

```
import copy
nested_list = [[1, 2], [3, 4]]
shallow_copy = nested_list.copy()
deep_copy = copy.deepcopy(nested_list)

nested_list[0].append(99)

print("Original:", nested_list)
print("Superficial:", shallow_copy)
print("Profunda:", deep_copy)
```

Responde:

1. ¿Por qué `shallow_copy[0]` contiene `99`?
2. ¿Por qué `deep_copy[0]` NO contiene `99`?
3. Dibuja el diagrama de memoria que explica la diferencia.

Criterio de éxito: la respuesta explica que la copia superficial copia el contenedor exterior pero ambas listas comparten las sublistas interiores. La copia profunda copia todo recursivamente.

Contexto Profesional

Rol: Analista de datos procesando un CSV de ventas
Pandas añade sus propias reglas sobre el almacenamiento de datos. En pandas 3, *Copy-on-Write* impide que una selección modifique accidentalmente el `DataFrame` de origen. Una columna seleccionada es otro objeto `Series`; no debes deducir por ello si comparte almacenamiento interno.
Si quieres modificar el `DataFrame`, hazlo explícitamente con `.loc`:

```
import pandas as pd

df = pd.DataFrame({"precio": [10, 20, 30], "cantidad": [1, 2, 3]})
df.loc[0, "precio"] = 999
```

Si quieres una serie independiente, pide una copia:

```
prices = df["precio"].copy()
prices.iloc[0] = 500
# df conserva 999 en su primera fila
```

En pandas 2 y versiones anteriores, ciertas asignaciones encadenadas podían producir `SettingWithCopyWarning` porque su resultado dependía de si una selección era vista o copia. Con pandas 3 esas asignaciones no actualizan el original y se señalan como `ChainedAssignmentError`. La regla estable es expresar la intención: `.loc` para modificar el `DataFrame`; `.copy()` para trabajar con un objeto independiente.

2.4. El modelo completo

Recapitulemos todo el modelo en una imagen mental unificada:

Contexto Profesional

Rol: Desarrollador revisando código de un compañero

Te llega un PR con este código:

```
def update_scores(players_dict):
    players_dict["alice"] = 100
    return players_dict

scores = {"alice": 50, "bob": 75}
update_scores(scores)
print(scores)  # ¿Qué imprime?
```

¿Tiene un bug? Según el modelo de etiquetas: `update_scores` recibe una etiqueta a un diccionario mutable. Modifica el diccionario original sin que la función devuelva nada específico. El diccionario `scores` sí cambia (imprime `{"alice": 100, "bob": 75}`).
¿Es eso un bug o un feature? Depende del contrato de la función. Si la función se llama `update_scores`, el lector espera que actualice. Si la función se llamara `calculate_new_scores`, el lector espera una copia. El modelo de etiquetas te ayuda a razonar sobre esto claramente: siempre que pases un mutable, recuerda que el llamador puede verlo modificado. Documéntalo o usa `.copy()`.

1. **Todo es un objeto.** En Python, los números, las cadenas, las funciones, las clases, incluso los módulos, son objetos. Tienen identidad, tipo y valor.
2. **Las variables son etiquetas.** No cajas. Una variable es un nombre que apunta a un objeto. Puedes pegar y despegar etiquetas a voluntad.
3. **El tipo está en el objeto.** Por eso no declaras tipos en Python: cuando preguntas `type(x)`, le preguntas al objeto, no a la variable.
4. **Algunos objetos son inmutables.** No se pueden modificar. Cualquier operación que parezca una modificación crea un objeto nuevo.
5. **Otros objetos son mutables.** Se pueden modificar en el sitio. Si dos etiquetas apuntan al mismo objeto mutable, las modificaciones se ven desde ambas.
6. **Identidad es distinta de igualdad.** `==` pregunta por el contenido, `is` pregunta por la identidad.

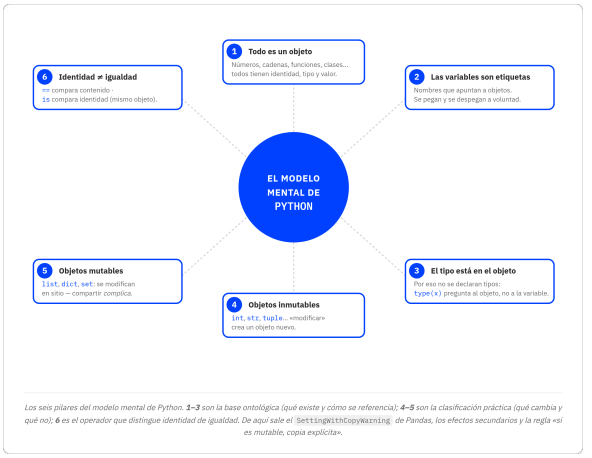

Figura 2.6.: El modelo completo de Python.

Tabla 2.5.: Tipos de Python clasificados por mutabilidad y seguridad al compartir entre variables.

Tipo	Mutable	Ejemplo	¿Compartir es seguro?
`int`	No	`42`	Sí
`float`	No	`3.14`	Sí
`str`	No	`"hola"`	Sí
`tuple`	No	`(1, 2, 3)`	Sí, si sus elementos también son inmutables
`list`	Sí	`[1, 2, 3]`	Depende del contrato; `.copy()` solo copia el contenedor
`dict`	Sí	`{"a": 1}`	Depende del contrato; `.copy()` solo copia el contenedor
`set`	Sí	`{1, 2, 3}`	Depende del contrato

Pro-Tip

Consejo exprés: si un objeto es mutable, consulta el contrato de la operación: ¿debe modificarlo o devolver otro? Si necesitas aislarlo, recuerda que `.copy()` es superficial y comprueba también los objetos anidados.

Manos a la Obra

Manos a la obra — Ejercicio 5: Detecta el efecto secundario
Para cada uno de los siguientes fragmentos, responde:

- ¿Tiene efecto secundario no deseado? (Sí/No)

- ¿Por qué?

Fragmento A:

```
def add_tax(prices):
    prices.append(0)
    return prices

price_list = [10, 20, 30]
result = add_tax(price_list)
print(price_list)  # ¿ha cambiado?
```

Fragmento B:

```
def get_first_element(items):
    return items[0]

my_list = [1, 2, 3]
value = get_first_element(my_list)
print(value)      # ¿qué imprime?
print(my_list)    # ¿ha cambiado?
```

Fragmento C:

```
def add_value(data, value):
    data["key"] = value
    return data

info = {"a": 1}
result = add_value(info, 2)
print(info)  # ¿ha cambiado?
```

Criterio de éxito: para cada fragmento indicas si tiene efecto secundario o no, y justificas la respuesta mencionando la mutabilidad del tipo involucrado.

2.5. Preguntas de reflexión

Pregunta de Reflexión

1. En Java, `int x = 5; int y = x; x = 6;` garantiza que `y` sigue siendo 5. Python da la misma garantía cuando `x` e `y` apuntan a enteros. ¿Y si `x` e `y` apuntan a una lista? Razona tu respuesta usando los conceptos de mutabilidad y etiquetas.

2. ¿Por qué Python decidió que los strings fueran inmutables? ¿Qué ventajas tiene esto para el rendimiento y la seguridad? Piensa en cómo se comportaría un diccionario si las claves pudieran modificarse.
3. Tienes una función que recibe una lista como argumento y necesita transformar sus valores. ¿Cómo decides si debes modificar la lista en el sitio o devolver una nueva lista? ¿Qué criterios usarías para elegir?
4. En pandas, ¿cuándo usarías `.loc` y cuándo `.copy()`? Explica por qué una selección es otro objeto Python aunque pandas pueda gestionar su almacenamiento internamente.

En resumen: las variables en Python no son cajas, son etiquetas. El tipo pertenece al objeto, no a la variable. Los objetos pueden ser mutables (se modifican en el sitio) o inmutables (su estructura no cambia). Dos operadores —`==` e `is`— distinguen entre igualdad de valor e identidad. Cuando necesites aislar un mutable, decide qué profundidad debes copiar.

Este modelo mental será la base de todo lo que viene. En el siguiente capítulo veremos los tipos básicos y los operadores; después llegarán las listas, los diccionarios y las funciones. En todos esos casos, las variables seguirán siendo etiquetas que apuntan a objetos.

Una última conexión: Pandas construye sus propias reglas sobre el modelo de objetos de Python. En pandas 3, *Copy-on-Write* evita que una selección cambie el `DataFrame` de origen. Usa `.loc` cuando quieras modificarlo y `.copy()` cuando necesites un objeto independiente.