

Tabla de contenidos

Prefacio	xv
Para quién es este libro	xv
Cómo está organizado	xv
Las cajas de colores	xvi
Sobre la terminología	xvii
Agradecimientos	xix
Configurar el entorno Python	1
Ruta A — Google Colab	2
Ruta B — Miniconda + VS Code	4
B.1 Instalar Miniconda	4
B.2 Crear el entorno del libro	5
B.3 Instalar y configurar VS Code	6
B.4 Verificar el entorno	7
Ruta C — Entorno local	9
Jupyter vs scripts .py	9
Preguntas de reflexión	10
I. El Cambio de Paradigma	11
1. El Laboratorio	13
1.1. Por qué Python para el análisis de datos	13
1.1.1. El stack del analista	14
1.1.2. Python cubre todo el ciclo	16
1.1.3. Adelanto de Capítulo 7: la potencia del filtrado en Python	17
1.2. El cuaderno como diario de investigación	18
1.2.1. Anatomía del cuaderno	18
1.2.2. ¿Por qué JSON?	20
1.3. Dominando el entorno	21
1.3.1. Atajos de teclado esenciales	21
1.3.2. Comandos mágicos	22
1.3.3. La terminal desde el cuaderno	22
1.3.4. Markdown esencial para el cuaderno	23
1.4. Reglas del profesional	24
1.4.1. El orden de ejecución importa: la regla de oro	24
1.4.2. Organización del espacio de trabajo	26
1.5. Preguntas de reflexión	28

2.	El Modelo Mental Correcto	29
2.1.	De cajas a etiquetas	29
2.1.1.	El diagrama mental	30
2.1.2.	Las dos herramientas de inspección	31
2.2.	Identidad vs. igualdad	32
2.2.1.	Demostración con listas	33
2.2.2.	El caso de uso principal: None	33
2.3.	Mutabilidad e inmutabilidad	35
2.3.1.	Tipos mutables vs. inmutables	35
2.3.2.	Demostración con inmutables	35
2.3.3.	Demostración con mutables	36
2.3.4.	La solución: copia explícita	37
2.3.5.	Copia superficial vs. copia profunda	38
2.4.	El modelo completo	40
2.5.	Preguntas de reflexión	43
II.	Los Datos	45
3.	Tipos y Operadores	47
3.1.	Sintaxis Python: la indentación como estructura	47
3.2.	Tipos escalares	49
3.2.1.	Enteros (int)	49
3.2.2.	Flotantes (float)	50
3.2.3.	Booleanos (bool)	51
3.2.4.	None	51
3.2.5.	Truthiness	52
3.3.	Operadores	54
3.3.1.	Aritméticos	54
3.3.2.	Lógicos	55
3.3.3.	Comparaciones encadenadas	55
3.3.4.	Identidad y pertenencia	56
3.4.	Cadenas de texto (str)	58
3.4.1.	Comillas	58
3.4.2.	f-strings (Python 3.6+)	58
3.4.3.	Métodos esenciales	59
3.4.4.	Slicing de strings	60
3.5.	Preguntas de reflexión	61
4.	Colecciones	63
4.1.	Listas	63
4.1.1.	Creación y acceso	63
4.1.2.	Métodos esenciales	64
4.1.3.	Ordenar: in-place vs. nueva lista	65
4.2.	Indexación y slicing: el superpoder de las secuencias	66
4.2.1.	Indexación básica	66
4.2.2.	Slicing: [inicio:fin]	67
4.2.3.	Slicing con paso: [inicio:fin:paso]	67
4.3.	Tuplas: inmutabilidad con propósito	69
4.3.1.	Cuándo usar una tupla	70

4.3.2.	Desempaquetado de tuplas	70
4.4.	Diccionarios: acceso O(1) por clave	71
4.4.1.	Comparativa: buscar en lista vs. buscar en diccionario	72
4.4.2.	Métodos esenciales	73
4.4.3.	Fusión de diccionarios	74
4.4.4.	Casos de uso en análisis de datos	74
4.5.	Sets: deduplicación y operaciones de conjuntos	75
4.5.1.	Deduplicación	75
4.5.2.	Operaciones de conjuntos	76
4.5.3.	Subconjuntos y superconjuntos	77
4.5.4.	Creación de sets a partir de otros tipos	77
4.5.5.	Conteo de frecuencias	78
4.6.	Resumen comparativo	79
4.7.	Preguntas de reflexión	80

III. Transformar Datos 81

5.	Funciones	83
5.1.	DRY: el principio que justifica las funciones	83
5.2.	Anatomía de una función	84
5.3.	Formas de pasar argumentos	87
5.3.1.	Argumentos posicionales	87
5.3.2.	Argumentos keyword	87
5.3.3.	Argumentos por defecto	87
5.4.	Scope: la regla LEGB	88
5.4.1.	El error más común: modificar una variable global	89
5.4.2.	Verifica tu predicción	91
5.5.	*args y **kwargs: argumentos variables	92
5.5.1.	*args: argumentos posicionales variables	92
5.5.2.	**kwargs: argumentos keyword variables	93
5.5.3.	Combinación	93
5.6.	Funciones lambda	94
5.6.1.	Uso principal: como argumento de otras funciones	95
5.6.2.	Cuándo usar lambda	95
5.7.	Funciones como objetos de primera clase	96
5.8.	Preguntas de reflexión	98
6.	Flujo Imperativo	101
6.1.	if / elif / else	101
6.1.1.	Truthiness en acción	102
6.1.2.	Operador ternario	102
6.2.	El bucle for — for-each, no for-contador	103
6.2.1.	enumerate(): índice Y valor	104
6.2.2.	zip(): iterar varias listas en paralelo	105
6.2.3.	break y continue	106
6.2.4.	for/else — peculiaridad pythónica	106
6.3.	while	108
6.3.1.	while True CON break	109
6.3.2.	¿for O while?	109

6.4.	La limitación del estilo imperativo	110
6.5.	Preguntas de reflexión	111
7.	Comprehensions: El Camino Funcional I	113
7.1.	De 3 líneas a 1: la list comprehension	113
7.1.1.	Anatomía	114
7.1.2.	Primeros ejemplos	115
7.1.3.	Comparativa directa: imperativo vs. comprehension	115
7.1.4.	Rendimiento: comprehension vs. bucle	116
7.1.5.	También funciona con strings	117
7.2.	El filtro vs. la expresión condicional	118
7.2.1.	if al final = FILTRO	119
7.2.2.	if/else al principio = EXPRESIÓN CONDICIONAL	119
7.3.	Dict comprehensions	120
7.3.1.	Dict comprehension con condición	122
7.4.	Set comprehensions	123
7.5.	Comprehensions anidadas	124
7.6.	Cuándo NO usar una comprehension	125
7.6.1.	Otros errores comunes	126
7.7.	Generator expressions (adelanto)	127
7.8.	Preguntas de reflexión	130
8.	Programación Funcional — El Camino Funcional II	131
8.1.	Funciones de orden superior	131
8.2.	map() — transformar cada elemento	132
8.2.1.	map() vs. comprehension	132
8.2.2.	map() con múltiples iterables	133
8.3.	filter() — seleccionar elementos	133
8.3.1.	filter() vs. comprehension	134
8.3.2.	Combinar map() y filter()	134
8.3.3.	filter() CON None	135
8.4.	functools.reduce() — plegar una secuencia	136
8.4.1.	Cómo funciona internamente	136
8.4.2.	Valor inicial	137
8.4.3.	¿Cuándo usar reduce()?	138
8.5.	functools.partial() — especializar funciones	139
8.5.1.	Caso de uso en análisis	140
8.6.	Encadenamiento: la tubería de datos	141
8.6.1.	El mismo problema, tres estilos	141
8.6.2.	La regla de la legibilidad	144
8.7.	Comparativa final: ¿cuándo usar qué?	144
8.8.	Preguntas de reflexión	146
IV.	Python Robusto	147
9.	Iteradores y Generadores	149
9.1.	El protocolo de iteración — iter() y next()	149
9.1.1.	Iterable vs. iterador	150
9.2.	Evaluación perezosa — por qué range(10**9) no explota	152

9.3.	enumerate(), zip() e itertools — iteradores útiles	154
9.3.1.	itertools — la navaja suiza de los iteradores	155
9.4.	Funciones generadoras — yield	157
9.4.1.	Fibonacci como caso paradigmático	158
9.5.	Generator expressions vs. list comprehensions	159
9.6.	¿Por qué importa? — Perspectiva de analista	161
9.6.1.	Tuberías de datos completamente perezosas	161
9.6.2.	Pandas lee en chunks	162
9.7.	Resumen del capítulo	162
10.	Python Profesional	165
10.1.	Excepciones — un programa robusto sabe fallar	165
10.1.1.	La estructura try/except	165
10.1.2.	Excepciones que todo analista debe conocer	166
10.1.3.	La mala práctica del except genérico	167
10.1.4.	raise para validar entradas	168
10.2.	Módulos — organizar el código	169
10.2.1.	Las tres formas de importar	170
10.2.2.	Crear tu propio módulo	171
10.3.	La librería estándar — sin instalar nada	172
10.4.	El ecosistema de data science — visión general	173
10.4.1.	Una muestra del poder combinado	175
10.4.2.	Ejercicios finales integrados	176
10.5.	Cierre — el camino que has recorrido	177
10.6.	Resumen del capítulo	178
	Apéndices	179
	Errores comunes para programadores Java y C++	179
	Cómo usar este apéndice	179
	Sintaxis	179
	SyntaxError: invalid syntax — falta el :	179
	TabError / IndentationError — mezcla de tabs y espacios	180
	elif en lugar de else if	180
	Variables y scope	181
	NameError: name 'x' is not defined	181
	UnboundLocalError: local variable referenced before assignment	181
	Tipos y operaciones	182
	TypeError: can only concatenate str (not "int") to str	182
	TypeError: 'NoneType' object is not subscriptable (o similar)	182
	Comparar con == en lugar de is para None	182
	Colecciones	183
	IndexError: list index out of range	183
	KeyError al acceder a un diccionario	183
	TypeError: unhashable type: 'list'	183
	Mutabilidad	184
	Alias inesperado: modificar una lista “copia”	184
	Argumento mutable por defecto	184

Funciones	185
for i in range(len(items)) cuando no necesitas el índice	185
Llamar a un método Java que no existe en Python	185
Módulos e imports	186
ModuleNotFoundError: No module named 'x'	186
ImportError — nombre incorrecto al importar	186
Encoding	186
UnicodeDecodeError al leer ficheros	186
Referencia rápida de excepciones	187
Equivalencias Java → Python en código	187
Lecturas y Recursos Recomendados	189
Para seguir con Python	189
Para programación funcional	189
Documentación oficial	190
Para análisis de datos	190
Para visualización y storytelling	190
Para entornos y despliegue	191
El libro complementario	191
Epílogo	193
Otros libros	195
Análisis de Datos con Python	195

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
```

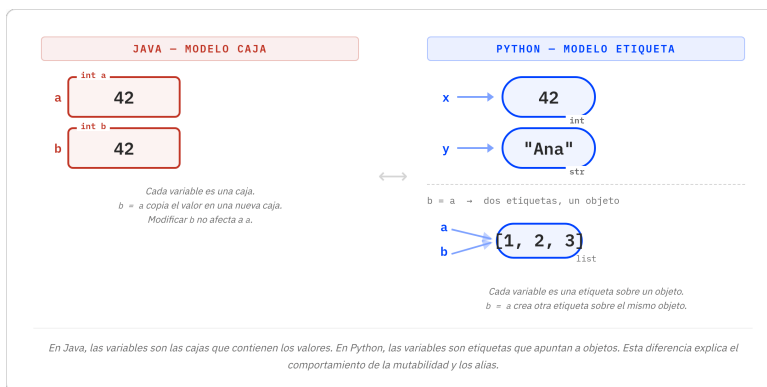


Figura 2.1.: Modelo de etiquetas de Python.

`x` y `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
<code>type(x)</code>	Tipo del objeto	<code>x.getClass()</code>
<code>id(x)</code>	Identificador de identidad durante la vida del objeto	<code>System.identityHashCode(x)</code> , 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.

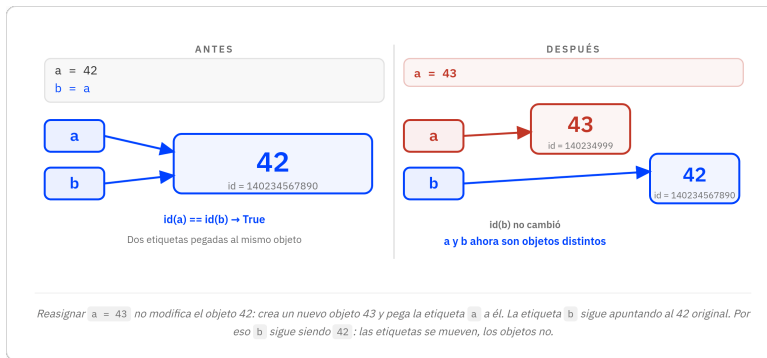


Figura 2.2.: Modelo de memoria.

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()
```

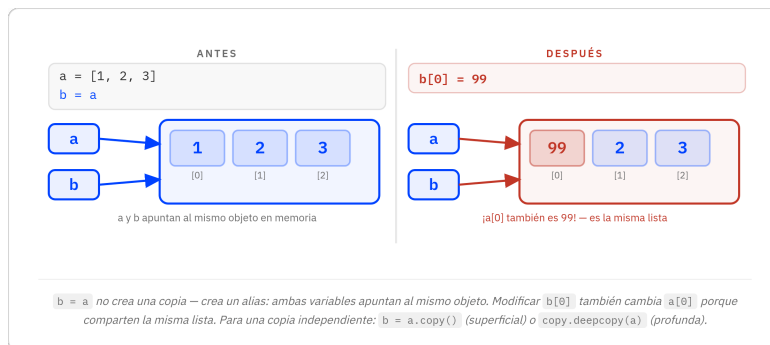


Figura 2.3.: Alias y mutación.

`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 $\approx$ `.equals()` en Java, y `is` en Python $\approx$ `==` 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
<code>x == y</code>		
<code>x is y</code>		
<code>x == z</code>		
<code>x is z</code>		
<code>x == w</code>		
<code>x is w</code>		

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
```

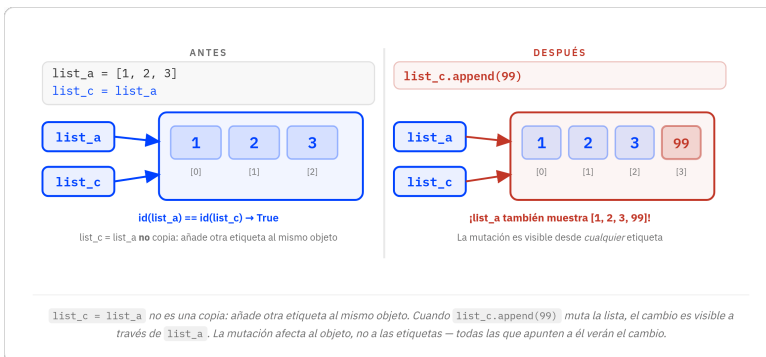


Figura 2.4.: Mutación compartida.

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.

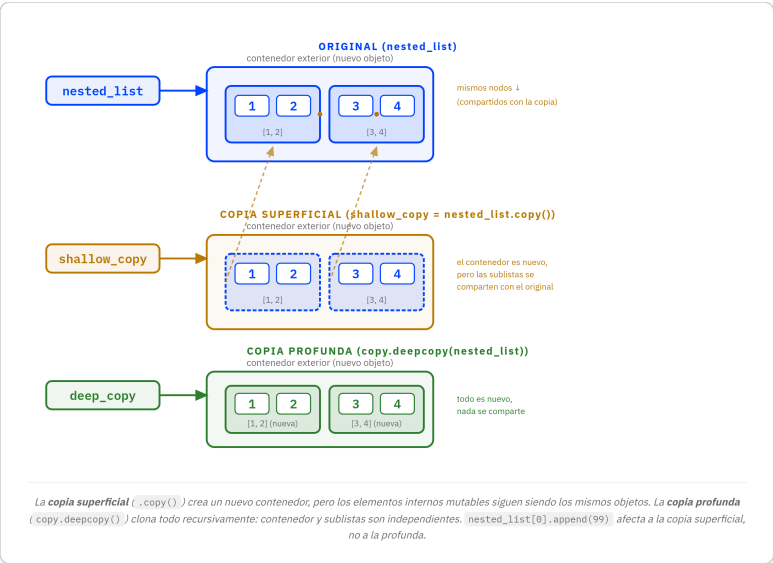


Figura 2.5.: Copia superficial vs. copia profunda.

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

Copia	Cómo se hace	Cuándo usarla
Superficial	<code>.copy()</code> , <code>list()</code> , <code>dict()</code>	Listas planas (elementos inmutables)
Profunda	<code>copy.deepcopy()</code>	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

Recapitemos 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. Documentalo 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.

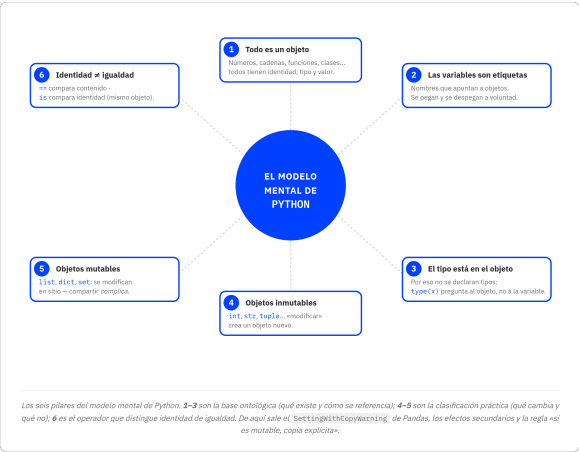


Figura 2.6.: El modelo completo de Python.

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; <code>.copy()</code> solo copia el contenedor
dict	Sí	{"a": 1}	Depende del contrato; <code>.copy()</code> 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.