对象引用、可变性

本章的主题是对象与对象名称之间的区别。名称不是对象，而是单独的东西。

变量和盒子

Python 变量类似于Java 中的引用式变量，因此最好把它们理解为附加在对象上的标注。

示例 8-1　变量 a 和 b 引用同一个列表，而不是那个列表的副本

```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> b
[1, 2, 3, 4]
```

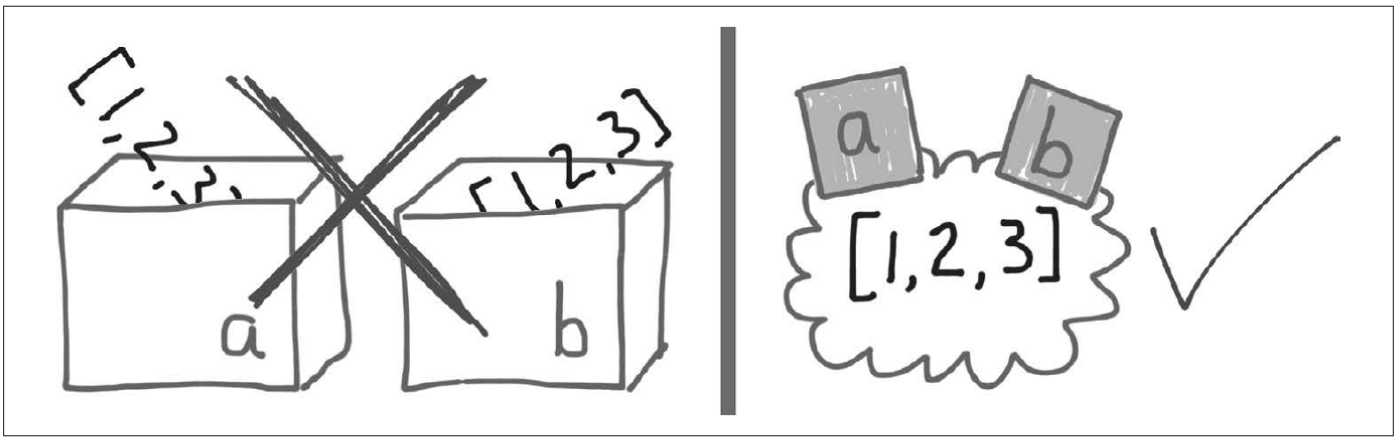

图 8-1：如果把变量想象为盒子，那么无法解释 Python 中的赋值；应该把变量视作便利贴，这样示例 8-1 中的行为就好解释了

为了理解Python 中的赋值语句，应该始终先读右边。对象在右边创建或获取，在此之后左边的变量才会绑定到对象上，这就像为对象贴上标注。创建对象之后才会把变量分配给对象，先有图中的盒子，才会有a和b指向它。

因为变量只不过是标注，所以无法阻止为对象贴上多个标注。贴的多个标注，就是别名。

标识、相等性和别名

先看下面的例子，理解别名。

示例 8-3　charles 和 lewis 指代同一个对象

```
>>> charles = {'name': 'Charles L. Dodgson', 'born': 1832}
>>> lewis = charles  ❶
>>> lewis is charles
True
>>> id(charles), id(lewis)  ❷
(4300473992, 4300473992)
>>> lewis['balance'] = 950  ❸
>>> charles
{'name': 'Charles L. Dodgson', 'balance': 950, 'born': 1832}
```

❶ lewis 是 charles 的别名。
❷ is 运算符和 id 函数确认了这一点。
❸ 向 lewis 中添加一个元素相当于向 charles 中添加一个元素。

is 判断指向的是不是同一个盒子，==只能判断指向的盒子中的内容是否一样。

示例 8-4　alex 与 charles 比较的结果是相等，但 alex 不是 charles

```
>>> alex = {'name': 'Charles L. Dodgson', 'born': 1832, 'balance': 950}  ❶
>>> alex == charles  ❷
True
>>> alex is not charles  ❸
True
```

更加形象的展示如下图所示。

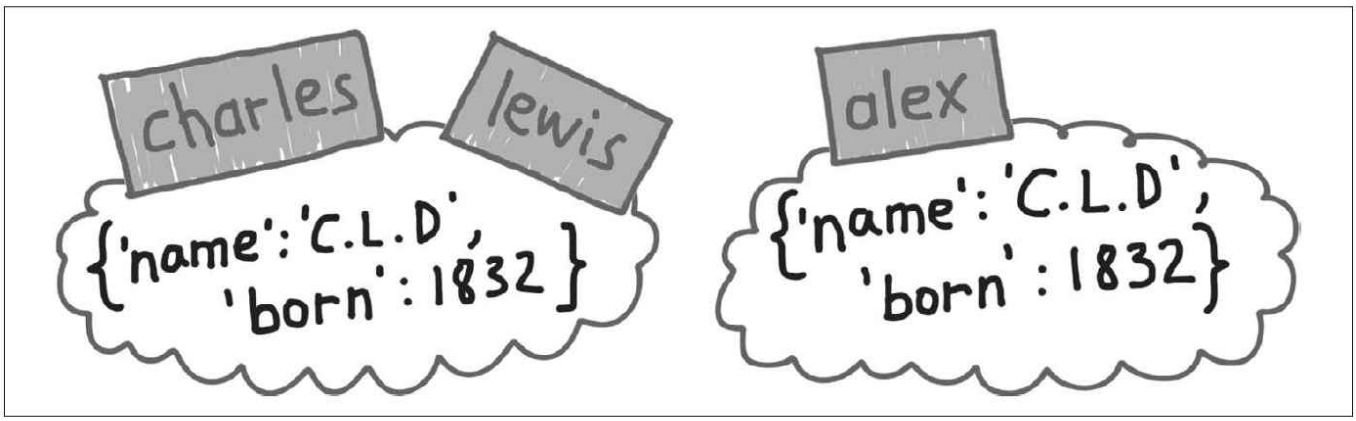

图 8-2：charles 和 lewis 绑定同一个对象，alex 绑定另一个具有相同内容的对象

对象一旦创建，它的标识绝不会变；你可以把标识理解为对象在内存中的地址。is 运算符比较两个对象的标识；id() 函数返回对象标识的整数表示。

元组的相对不可变性

元组与多数Python 集合（列表、字典、集，等等）一样，保存的是对象的引用（str、bytes array.array等单一类型序列是扁平的，它们保存的不是引用，而是在连续的内存中保存数据本身（字符、字节和数字））。如果引用的元素是可变的，即便元组本身不可变，元素依然可变。元组的值会随着引用的可变对象的变化而变。元组中不可变的是元素的标识。

```
>>> t1 = (1, 2, [30, 40]) # t1不可变，但t1[-1]可变。
>>> t2 = (1, 2, [30, 40]) # t2内容和t1相同，但是id不一样，因为它们指向不同的对象
>>> t1 == t2 # 虽然t1和t2是不同的对象，但是二者相等——与预期相符。
True
>>> id(t1[-1])
4302515784
>>> t1[-1].append(99) # 就地修改t1[-1] 列表。
>>> t1
(1, 2, [30, 40, 99])
>>> id(t1[-1]) # t[-1]的地址没变，但是内容变了
4302515784
>>> t1 == t2 # 现在，t1和t2内容不一样了，t1 和t2 不相等。
False
```

什么是可散列的数据类型？这也是有些元组不可散列的原因。元组的相对不可变性解释了2.6.1 节的谜题。

复制对象的时候，副本和原来对象的值相等，但是id不同。但是如果这个对象中含有其他对象，那么会复制内部对象吗？可以共享内部对象吗？这些问题没有唯一的答案。继续往下看。

浅复制

复制列表（或多数内置的可变集合）最简单的方式是使用内置的类型构造方法。

```
>>> l1 = [3, [55, 44], (7, 8, 9)]
>>> l2 = list(l1) # list(l1) 创建l1 的副本。
>>> l2
[3, [55, 44], (7, 8, 9)]
>>> l2 == l1 # 副本与源列表相等。
True
>>> l2 is l1 # 但是二者指代不同的对象。
False
```

对列表和其他可变序列来说，还能使用简洁 `l2 = l1[:]` 语句创建副本。构造方法或 `[:]` 做的是浅复制（即复制了最外层容器，副本中的元素是源容器中元素的引用）。

如果所有元素都是不可变的，那么这样没有问题，还能节省内存。但是，如果有可变的元素，可能就会导致意想不到的问题。

为一个包含另一个列表的列表做浅复制：

```
l1 = [3, [66, 55, 44], (7, 8, 9)]
```

```
l2 = list(l1) # l2 是l1 的浅复制副本，此时的状态如图8-3 所示。
```

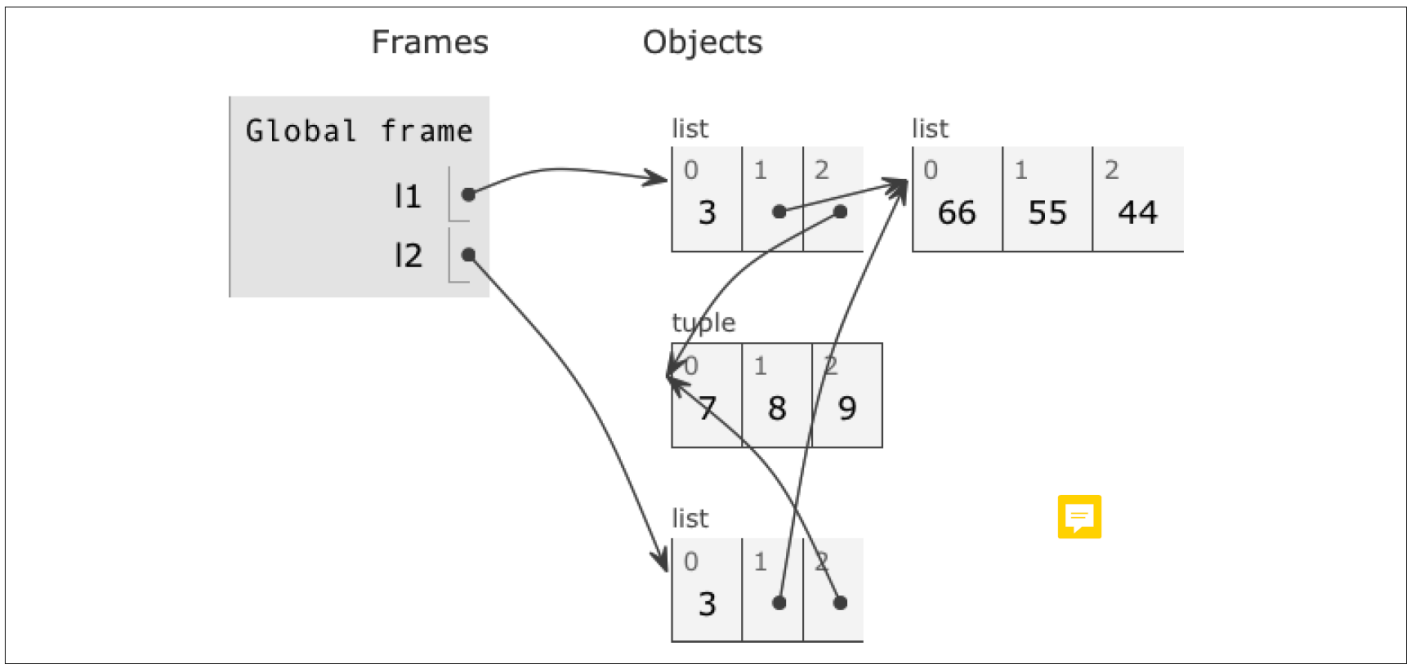

图 8-3：示例 8-6 执行 l2 = list(l1) 赋值后的程序状态。l1 和 l2 指代不同的列表，但是二者引用同一个列表 [66，55，44] 和元组（7，8，9）（图表由 Python Tutor 网站生成）

list和tuple保存的是对象的引用，太直观了！由此可以看出来，创建l2的时候，只是新建了一个3，列表[66,55,44]和元组(7,8,9)都没有新建。

```
l1.append(100) # 把100 追加到l1 中，对l2 没有影响。
l1[1].remove(55) # 把内部列表l1[1] 中的55 删除。这对l2 有影响，因为l2[1] 绑定的列表与l1[1] 是同一个。
print('l1:', l1)
print('l2:', l2)
# 输出
l1: [3, [66, 44], (7, 8, 9), 100]
l2: [3, [66, 44], (7, 8, 9)]

l2[1] += [33, 22] # 对可变的对象来说，如l2[1] 引用的列表，+= 运算符就地修改列表。这次修改在l1[1]中也有体现，因为它是l2[1] 的别名。
l2[2] += (10, 11) # 对元组来说，+= 运算符创建一个新元组，然后重新绑定给变量l2[2]。这等同于l2[2] =l2[2] + (10, 11)。现在，l1 和l2 中最后位置上的元组不是同一个对象。如图8-4 所示。
print('l1:', l1)
print('l2:', l2)
# 输出
l1: [3, [66, 44, 33, 22], (7, 8, 9), 100]
l2: [3, [66, 44, 33, 22], (7, 8, 9, 10, 11)]
```

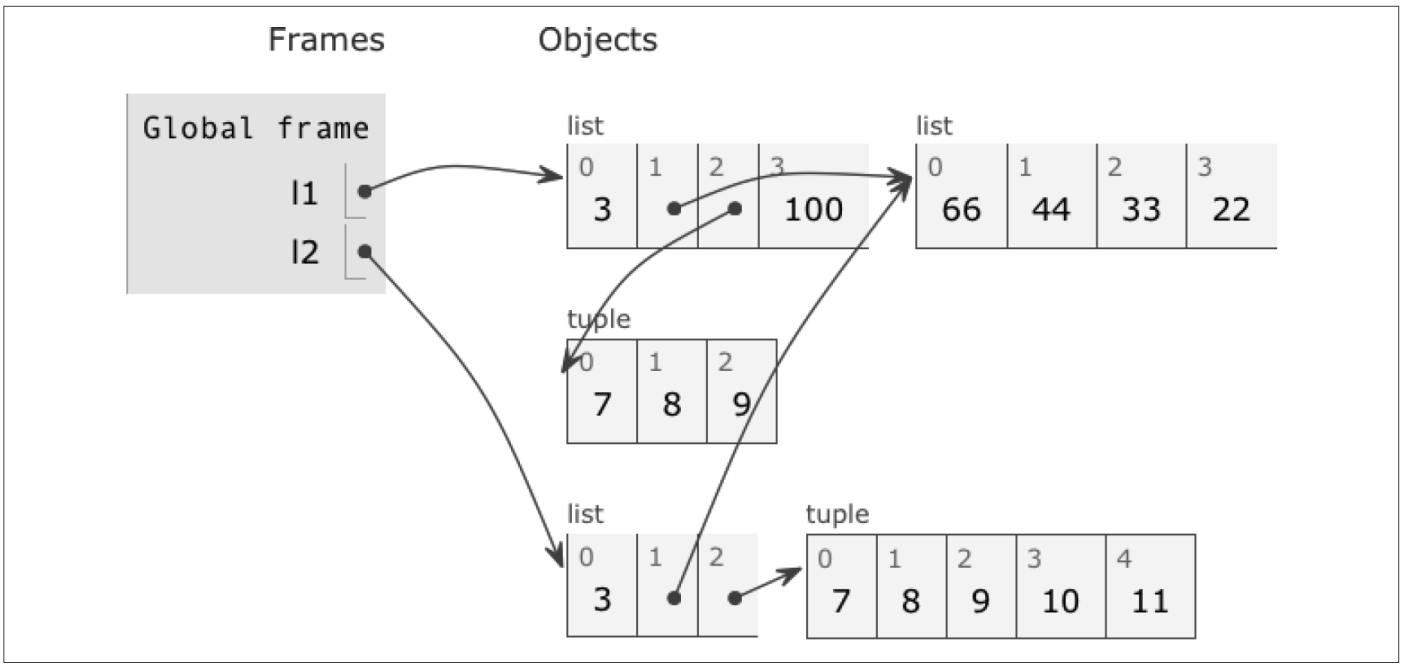

图 8-4：l1 和 l2 的最终状态：二者依然引用同一个列表对象，现在列表的值是 [66, 44, 33, 22]，不过 l2[2] += (10, 11) 创建一个新元组，内容是 (7, 8, 9, 10, 11)，它与 l1[2] 引用的元组 (7, 8, 9) 无关（图表由 Python Tutor 网站生成）

问题就出在，l1和l2指向了同一个list，而这个list改变的时候是自增的，不会新建，而元组会新建一个。

创建副本VS创建别名

这样会创建一个别名，修改会互相影响。

```
class TwilightBus:
    """让乘客销声匿迹的校车"""

    def __init__(self, passengers=None):
        if passengers is None:
            self.passengers = []  ❶
        else:
            self.passengers = passengers  ❷
```

如果不想创建一个别名，而是只希望把值复制过去，应该这样做。

```
def __init__(self, passengers=None):
    if passengers is None:
        self.passengers = []
    else:
        self.passengers = list(passengers)  ❶
```

函数的参数传递

Python 唯一支持的参数传递模式是共享传参（call by sharing）。多数面向对象语言都采用这一模式，包括Ruby、Smalltalk 和Java（Java 的引用类型是这样，基本类型按值传参）。

共享传参指函数的各个形式参数获得实参中各个引用的副本。也就是说，函数内部的形参是实参的别名。list是可变对象，所以下边的list变了。但是整数和元组是不可变的，所以没有被改变。

函数可能会修改接收到的任何可变对象

```
>>> def f(a, b):
... a += b
... return a
...
>>> x = 1
>>> y = 2
>>> f(x, y)
3
>>> x, y # 数字x没变
(1, 2)
>>> a = [1, 2]
>>> b = [3, 4]
>>> f(a, b)
[1, 2, 3, 4]
>>> a, b  # 列表a变了
([1, 2, 3, 4], [3, 4])
>>> t = (10, 20)
>>> u = (30, 40)
>>> f(t, u)
(10, 20, 30, 40)
>>> t, u # 元组t没变
((10, 20), (30, 40))
```

事实上，这两个参数不一定非要是a，b，换成别的也行，a的值还是会改变，因为传递过去的是别名，a代表的盒子的内容还是会改变。

```python
def g(a, b):
    a += b
    return a

x = [1, 2]
y = [3, 4]

print(g(x, y))
print(x, y)

"""
输出:
[1, 2, 3, 4]
[1, 2, 3, 4] [3, 4]
"""
```