CHAPTER 6

Object References, Mutability, and Recycling

> "You are sad," the Knight said in an anxious tone: "let me sing you a song to comfort you. […] The name of the song is called 'HADDOCKS' EYES'."
>
> "Oh, that's the name of the song, is it?" Alice said, trying to feel interested.
>
> "No, you don't understand," the Knight said, looking a little vexed. "That's what the name is CALLED. The name really IS 'THE AGED AGED MAN.'"
>
> —Adapted from Lewis Carroll, *Through the Looking-Glass, and What Alice Found There*

Alice and the Knight set the tone of what we will see in this chapter. The theme is the distinction between objects and their names. A name is not the object; a name is a separate thing.

We start the chapter by presenting a metaphor for variables in Python: variables are labels, not boxes. If reference variables are old news to you, the analogy may still be handy if you need to explain aliasing issues to others.

We then discuss the concepts of object identity, value, and aliasing. A surprising trait of tuples is revealed: they are immutable but their values may change. This leads to a discussion of shallow and deep copies. References and function parameters are our next theme: the problem with mutable parameter defaults and the safe handling of mutable arguments passed by clients of our functions.

The last sections of the chapter cover garbage collection, the `del` command, and a selection of tricks that Python plays with immutable objects.

This is a rather dry chapter, but its topics lie at the heart of many subtle bugs in real Python programs.

What's New in This Chapter

The topics covered here are very fundamental and stable. There were no changes worth mentioning in this second edition.

I added an example of using `is` to test for a sentinel object, and a warning about misuses of the `is` operator at the end of "Choosing Between == and is" on page 206.

This chapter used to be in Part IV, but I decided to bring it up earlier because it works better as an ending to Part II, "Data Structures," than an opening to "Object-Oriented Idioms."

The section on "Weak References" from the first edition of this book is now a post at *fluentpython.com*.

Let's start by unlearning that a variable is like a box where you store data.

Variables Are Not Boxes

In 1997, I took a summer course on Java at MIT. The professor, Lynn Stein,[1] made the point that the usual "variables as boxes" metaphor actually hinders the understanding of reference variables in object-oriented languages. Python variables are like reference variables in Java; a better metaphor is to think of variables as labels with names attached to objects. The next example and figure will help you understand why.

Example 6-1 is a simple interaction that the "variables as boxes" idea cannot explain. Figure 6-1 illustrates why the box metaphor is wrong for Python, while sticky notes provide a helpful picture of how variables actually work.

Example 6-1. Variables `a` and `b` hold references to the same list, not copies of the list

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

1 Lynn Andrea Stein is an award-winning computer science educator who currently teaches at Olin College of Engineering.

❶ Create a list `[1, 2, 3]` and bind the variable `a` to it.

❷ Bind the variable `b` to the same value that `a` is referencing.

❸ Modify the list referenced by `a`, by appending another item.

❹ You can see the effect via the `b` variable. If we think of `b` as a box that stored a copy of the `[1, 2, 3]` from the `a` box, this behavior makes no sense.

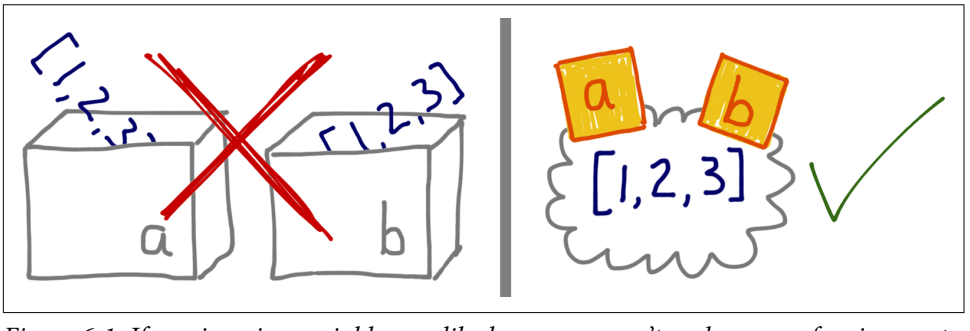

Figure 6-1. If you imagine variables are like boxes, you can't make sense of assignment in Python; instead, think of variables as sticky notes, and Example 6-1 becomes easy to explain.

Therefore, the `b = a` statement does not copy the contents of box `a` into box `b`. It attaches the label `b` to the object that already has the label `a`.

Prof. Stein also spoke about assignment in a very deliberate way. For example, when talking about a seesaw object in a simulation, she would say: "Variable *s* is assigned to the seesaw," but never "The seesaw is assigned to variable *s*." With reference variables, it makes much more sense to say that the variable is assigned to an object, and not the other way around. After all, the object is created before the assignment. Example 6-2 proves that the righthand side of an assignment happens first.

Since the verb "to assign" is used in contradictory ways, a useful alternative is "to bind": Python's assignment statement `x = …` binds the `x` name to the object created or referenced on the righthand side. And the object must exist before a name can be bound to it, as Example 6-2 proves.

Example 6-2. Variables are bound to objects only after the objects are created

```
>>> class Gizmo:
...     def __init__(self):
...         print(f'Gizmo id: {id(self)}')
...
>>> x = Gizmo()
```

```
Gizmo id: 4301489152  ❶
>>> y = Gizmo() * 10  ❷
Gizmo id: 4301489432  ❸
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for *: 'Gizmo' and 'int'
>>>
>>> dir()  ❹
['Gizmo', '__builtins__', '__doc__', '__loader__', '__name__',
'__package__', '__spec__', 'x']
```

❶ The output `Gizmo id:` … is a side effect of creating a `Gizmo` instance.

❷ Multiplying a `Gizmo` instance will raise an exception.

❸ Here is proof that a second `Gizmo` was actually instantiated before the multiplication was attempted.

❹ But variable `y` was never created, because the exception happened while the righthand side of the assignment was being evaluated.

To understand an assignment in Python, read the righthand side first: that's where the object is created or retrieved. After that, the variable on the left is bound to the object, like a label stuck to it. Just forget about the boxes.

Because variables are mere labels, nothing prevents an object from having several labels assigned to it. When that happens, you have *aliasing*, our next topic.

Identity, Equality, and Aliases

Lewis Carroll is the pen name of Prof. Charles Lutwidge Dodgson. Mr. Carroll is not only equal to Prof. Dodgson, they are one and the same. Example 6-3 expresses this idea in Python.

Example 6-3. `charles` and `lewis` refer to the same object

```
>>> 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', 'born': 1832, 'balance': 950}
```

❶ `lewis` is an alias for `charles`.

❷ The `is` operator and the `id` function confirm it.

❸ Adding an item to `lewis` is the same as adding an item to `charles`.

However, suppose an impostor—let's call him Dr. Alexander Pedachenko—claims he is Charles L. Dodgson, born in 1832. His credentials may be the same, but Dr. Pedachenko is not Prof. Dodgson. Figure 6-2 illustrates this scenario.

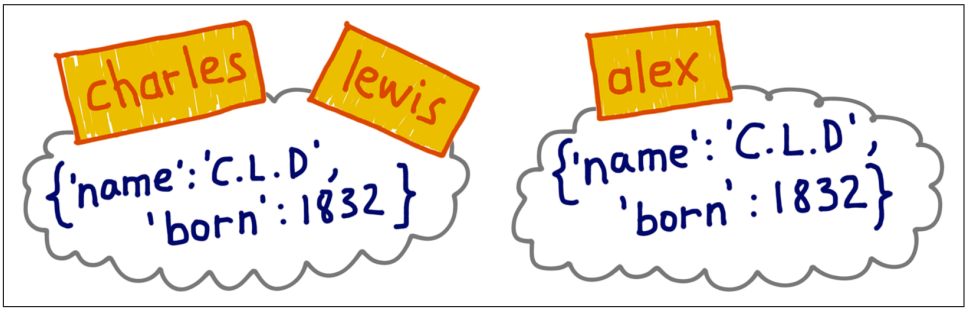

Figure 6-2. `charles` and `lewis` are bound to the same object; `alex` is bound to a separate object of equal value.

Example 6-4 implements and tests the `alex` object depicted in Figure 6-2.

Example 6-4. `alex` and `charles` compare equal, but `alex` is not `charles`

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

❶ `alex` refers to an object that is a replica of the object assigned to `charles`.

❷ The objects compare equal because of the `__eq__` implementation in the `dict` class.

❸ But they are distinct objects. This is the Pythonic way of writing the negative identity comparison: `a is not b`.

Example 6-3 is an example of *aliasing*. In that code, `lewis` and `charles` are aliases: two variables bound to the same object. On the other hand, `alex` is not an alias for

`charles`: these variables are bound to distinct objects. The objects bound to `alex` and `charles` have the same *value*—that's what `==` compares—but they have different identities.

In *The Python Language Reference*, "3.1. Objects, values and types" states:

> An object's identity never changes once it has been created; you may think of it as the object's address in memory. The `is` operator compares the identity of two objects; the `id()` function returns an integer representing its identity.

The real meaning of an object's ID is implementation dependent. In CPython, `id()` returns the memory address of the object, but it may be something else in another Python interpreter. The key point is that the ID is guaranteed to be a unique integer label, and it will never change during the life of the object.

In practice, we rarely use the `id()` function while programming. Identity checks are most often done with the `is` operator, which compares the object IDs, so our code doesn't need to call `id()` explicitly. Next, we'll talk about `is` versus `==`.

For tech reviewer Leonardo Rochael, the most frequent use for `id()` is while debugging, when the `repr()` of two objects look alike, but you need to understand whether two references are aliases or point to separate objects. If the references are in different contexts—such as different stack frames—using the `is` operator may not be viable.

Choosing Between == and is

The `==` operator compares the values of objects (the data they hold), while `is` compares their identities.

While programming, we often care more about values than object identities, so `==` appears more frequently than `is` in Python code.

However, if you are comparing a variable to a singleton, then it makes sense to use `is`. By far, the most common case is checking whether a variable is bound to `None`. This is the recommended way to do it:

```
x is None
```

And the proper way to write its negation is:

```
x is not None
```

`None` is the most common singleton we test with `is`. Sentinel objects are another example of singletons we test with `is`. Here is one way to create and test a sentinel object:

```
END_OF_DATA = object()
# ... many lines
def traverse(...):
    # ... more lines
    if node is END_OF_DATA:
        return
    # etc.
```

The `is` operator is faster than `==`, because it cannot be overloaded, so Python does not have to find and invoke special methods to evaluate it, and computing is as simple as comparing two integer IDs. In contrast, `a == b` is syntactic sugar for `a.__eq__(b)`. The `__eq__` method inherited from `object` compares object IDs, so it produces the same result as `is`. But most built-in types override `__eq__` with more meaningful implementations that actually take into account the values of the object attributes. Equality may involve a lot of processing—for example, when comparing large collections or deeply nested structures.

Usually we are more interested in object equality than identity. Checking for `None` is the *only* common use case for the `is` operator. Most other uses I see while reviewing code are wrong. If you are not sure, use `==`. It's usually what you want, and also works with `None`—albeit not as fast.

To wrap up this discussion of identity versus equality, we'll see that the famously immutable `tuple` is not as unchanging as you may expect.

The Relative Immutability of Tuples

Tuples, like most Python collections—lists, dicts, sets, etc.—are containers: they hold references to objects.[2] If the referenced items are mutable, they may change even if the tuple itself does not. In other words, the immutability of tuples really refers to the physical contents of the `tuple` data structure (i.e., the references it holds), and does not extend to the referenced objects.

Example 6-5 illustrates the situation in which the value of a tuple changes as a result of changes to a mutable object referenced in it. What can never change in a tuple is the identity of the items it contains.

2 In contrast, flat sequences like `str`, `bytes`, and `array.array` don't contain references but directly hold their contents—characters, bytes, and numbers—in contiguous memory.

Example 6-5. `t1` and `t2` initially compare equal, but changing a mutable item inside tuple `t1` makes it different

```
>>> t1 = (1, 2, [30, 40])  ❶
>>> t2 = (1, 2, [30, 40])  ❷
>>> t1 == t2  ❸
True
>>> id(t1[-1])  ❹
4302515784
>>> t1[-1].append(99)  ❺
>>> t1
(1, 2, [30, 40, 99])
>>> id(t1[-1])  ❻
4302515784
>>> t1 == t2  ❼
False
```

❶ `t1` is immutable, but `t1[-1]` is mutable.

❷ Build a tuple `t2` whose items are equal to those of `t1`.

❸ Although distinct objects, `t1` and `t2` compare equal, as expected.

❹ Inspect the identity of the list at `t1[-1]`.

❺ Modify the `t1[-1]` list in place.

❻ The identity of `t1[-1]` has not changed, only its value.

❼ `t1` and `t2` are now different.

This relative immutability of tuples is behind the riddle "A += Assignment Puzzler" on page 54. It's also the reason why some tuples are unhashable, as we've seen in "What Is Hashable" on page 84.

The distinction between equality and identity has further implications when you need to copy an object. A copy is an equal object with a different ID. But if an object contains other objects, should the copy also duplicate the inner objects, or is it OK to share them? There's no single answer. Read on for a discussion.

Copies Are Shallow by Default

The easiest way to copy a list (or most built-in mutable collections) is to use the built-in constructor for the type itself. For example:

```
>>> l1 = [3, [55, 44], (7, 8, 9)]
>>> l2 = list(l1)  ❶
```

```
>>> l2
[3, [55, 44], (7, 8, 9)]
>>> l2 == l1  ❷
True
>>> l2 is l1  ❸
False
```

❶ `list(l1)` creates a copy of `l1`.

❷ The copies are equal…

❸ …but refer to two different objects.

For lists and other mutable sequences, the shortcut `l2 = l1[:]` also makes a copy.

However, using the constructor or `[:]` produces a *shallow copy* (i.e., the outermost container is duplicated, but the copy is filled with references to the same items held by the original container). This saves memory and causes no problems if all the items are immutable. But if there are mutable items, this may lead to unpleasant surprises.

In Example 6-6, we create a shallow copy of a list containing another list and a tuple, and then make changes to see how they affect the referenced objects.

If you have a connected computer on hand, I highly recommend watching the interactive animation for Example 6-6 at the Online Python Tutor. As I write this, direct linking to a prepared example at *pythontutor.com* is not working reliably, but the tool is awesome, so taking the time to copy and paste the code is worthwhile.

Example 6-6. Making a shallow copy of a list containing another list; copy and paste this code to see it animated at the Online Python Tutor

```
l1 = [3, [66, 55, 44], (7, 8, 9)]
l2 = list(l1)       ❶
l1.append(100)      ❷
l1[1].remove(55)    ❸
print('l1:', l1)
print('l2:', l2)
l2[1] += [33, 22]   ❹
l2[2] += (10, 11)   ❺
print('l1:', l1)
print('l2:', l2)
```

❶ `l2` is a shallow copy of `l1`. This state is depicted in Figure 6-3.

❷ Appending `100` to `l1` has no effect on `l2`.

❸ Here we remove `55` from the inner list `l1[1]`. This affects `l2` because `l2[1]` is bound to the same list as `l1[1]`.

❹ For a mutable object like the list referred by `l2[1]`, the operator `+=` changes the list in place. This change is visible at `l1[1]`, which is an alias for `l2[1]`.

❺ `+=` on a tuple creates a new tuple and rebinds the variable `l2[2]` here. This is the same as doing `l2[2] = l2[2] + (10, 11)`. Now the tuples in the last position of `l1` and `l2` are no longer the same object. See Figure 6-4.

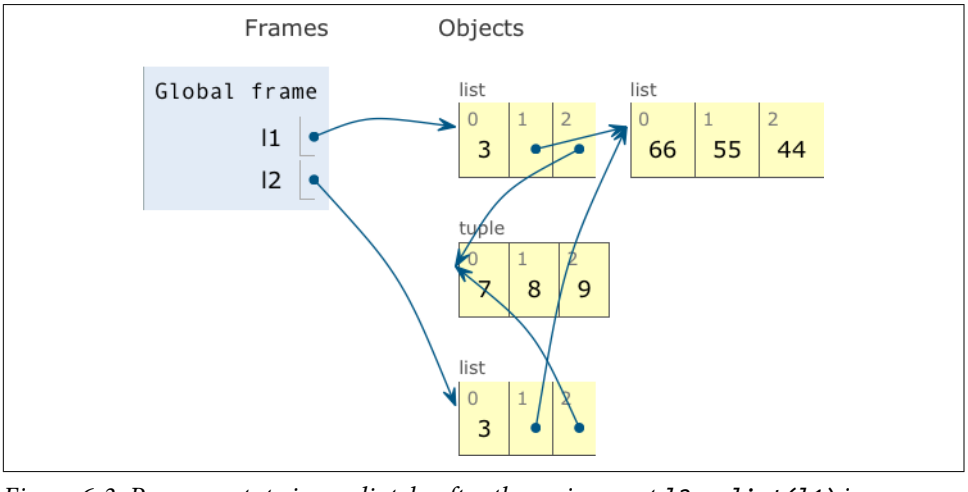

Figure 6-3. Program state immediately after the assignment `l2 = list(l1)` in Example 6-6. `l1` and `l2` refer to distinct lists, but the lists share references to the same inner list object `[66, 55, 44]` and tuple `(7, 8, 9)`. (Diagram generated by the Online Python Tutor.)

The output of Example 6-6 is Example 6-7, and the final state of the objects is depicted in Figure 6-4.

Example 6-7. Output of Example 6-6

```
l1: [3, [66, 44], (7, 8, 9), 100]
l2: [3, [66, 44], (7, 8, 9)]
l1: [3, [66, 44, 33, 22], (7, 8, 9), 100]
l2: [3, [66, 44, 33, 22], (7, 8, 9, 10, 11)]
```

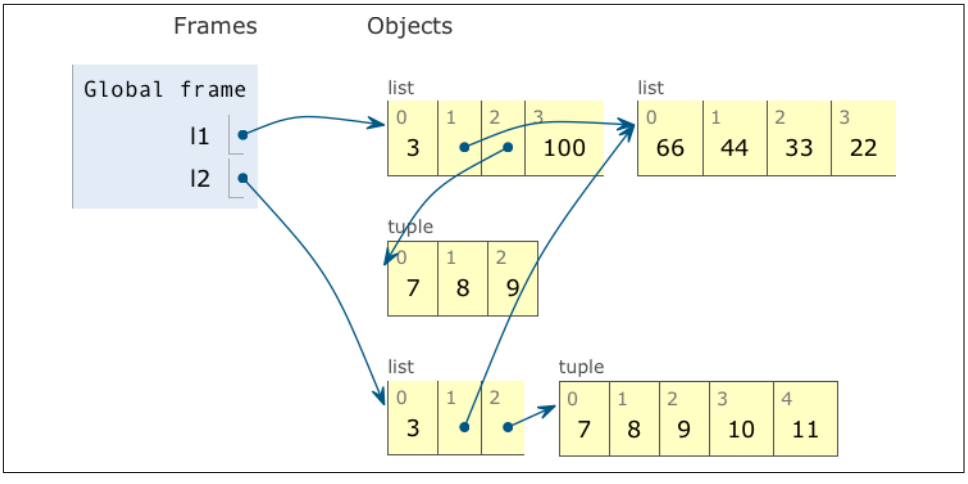

Figure 6-4. Final state of `l1` and `l2`: they still share references to the same list object, now containing `[66, 44, 33, 22]`, but the operation `l2[2] += (10, 11)` created a new tuple with content `(7, 8, 9, 10, 11)`, unrelated to the tuple `(7, 8, 9)` referenced by `l1[2]`. (Diagram generated by the Online Python Tutor.)

It should be clear now that shallow copies are easy to make, but they may or may not be what you want. How to make deep copies is our next topic.

Deep and Shallow Copies of Arbitrary Objects

Working with shallow copies is not always a problem, but sometimes you need to make deep copies (i.e., duplicates that do not share references of embedded objects). The `copy` module provides the `deepcopy` and `copy` functions that return deep and shallow copies of arbitrary objects.

To illustrate the use of `copy()` and `deepcopy()`, Example 6-8 defines a simple class, `Bus`, representing a school bus that is loaded with passengers and then picks up or drops off passengers on its route.

Example 6-8. Bus picks up and drops off passengers

```
class Bus:

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

    def pick(self, name):
        self.passengers.append(name)
```

```
    def drop(self, name):
        self.passengers.remove(name)
```

Now, in the interactive Example 6-9, we will create a `bus` object (`bus1`) and two clones—a shallow copy (`bus2`) and a deep copy (`bus3`)—to observe what happens as `bus1` drops off a student.

Example 6-9. Effects of using `copy` versus `deepcopy`

```
>>> import copy
>>> bus1 = Bus(['Alice', 'Bill', 'Claire', 'David'])
>>> bus2 = copy.copy(bus1)
>>> bus3 = copy.deepcopy(bus1)
>>> id(bus1), id(bus2), id(bus3)
(4301498296, 4301499416, 4301499752)  ❶
>>> bus1.drop('Bill')
>>> bus2.passengers
['Alice', 'Claire', 'David']          ❷
>>> id(bus1.passengers), id(bus2.passengers), id(bus3.passengers)
(4302658568, 4302658568, 4302657800)  ❸
>>> bus3.passengers
['Alice', 'Bill', 'Claire', 'David']  ❹
```

❶ Using `copy` and `deepcopy`, we create three distinct `Bus` instances.

❷ After `bus1` drops `'Bill'`, he is also missing from `bus2`.

❸ Inspection of the `passengers` attributes shows that `bus1` and `bus2` share the same list object, because `bus2` is a shallow copy of `bus1`.

❹ `bus3` is a deep copy of `bus1`, so its `passengers` attribute refers to another list.

Note that making deep copies is not a simple matter in the general case. Objects may have cyclic references that would cause a naïve algorithm to enter an infinite loop. The `deepcopy` function remembers the objects already copied to handle cyclic references gracefully. This is demonstrated in Example 6-10.

Example 6-10. Cyclic references: `b` refers to `a`, and then is appended to `a`; `deepcopy` still manages to copy `a`

```
>>> a = [10, 20]
>>> b = [a, 30]
>>> a.append(b)
>>> a
[10, 20, [[...], 30]]
>>> from copy import deepcopy
```

```
>>> c = deepcopy(a)
>>> c
[10, 20, [[...], 30]]
```

Also, a deep copy may be too deep in some cases. For example, objects may refer to external resources or singletons that should not be copied. You can control the behavior of both `copy` and `deepcopy` by implementing the `__copy__()` and `__deepcopy__()` special methods, as described in the copy module documentation.

The sharing of objects through aliases also explains how parameter passing works in Python, and the problem of using mutable types as parameter defaults. These issues will be covered next.

Function Parameters as References

The only mode of parameter passing in Python is *call by sharing*. That is the same mode used in most object-oriented languages, including JavaScript, Ruby, and Java (this applies to Java reference types; primitive types use call by value). Call by sharing means that each formal parameter of the function gets a copy of each reference in the arguments. In other words, the parameters inside the function become aliases of the actual arguments.

The result of this scheme is that a function may change any mutable object passed as a parameter, but it cannot change the identity of those objects (i.e., it cannot altogether replace an object with another). Example 6-11 shows a simple function using `+=` on one of its parameters. As we pass numbers, lists, and tuples to the function, the actual arguments passed are affected in different ways.

Example 6-11. A function may change any mutable object it receives

```
>>> def f(a, b):
...     a += b
...     return a
...
>>> x = 1
>>> y = 2
>>> f(x, y)
3
>>> x, y  ❶
(1, 2)
>>> a = [1, 2]
>>> b = [3, 4]
>>> f(a, b)
[1, 2, 3, 4]
>>> a, b  ❷
([1, 2, 3, 4], [3, 4])
>>> t = (10, 20)
```

```
>>> u = (30, 40)
>>> f(t, u)  ❸
(10, 20, 30, 40)
>>> t, u
((10, 20), (30, 40))
```

❶ The number `x` is unchanged.

❷ The list `a` is changed.

❸ The tuple `t` is unchanged.

Another issue related to function parameters is the use of mutable values for defaults, as discussed next.

Mutable Types as Parameter Defaults: Bad Idea

Optional parameters with default values are a great feature of Python function definitions, allowing our APIs to evolve while remaining backward compatible. However, you should avoid mutable objects as default values for parameters.

To illustrate this point, in Example 6-12, we take the `Bus` class from Example 6-8 and change its `__init__` method to create `HauntedBus`. Here we tried to be clever, and instead of having a default value of `passengers=None`, we have `passengers=[]`, thus avoiding the `if` in the previous `__init__`. This "cleverness" gets us into trouble.

Example 6-12. A simple class to illustrate the danger of a mutable default

```
class HauntedBus:
    """A bus model haunted by ghost passengers"""

    def __init__(self, passengers=[]):  ❶
        self.passengers = passengers  ❷

    def pick(self, name):
        self.passengers.append(name)  ❸

    def drop(self, name):
        self.passengers.remove(name)
```

❶ When the `passengers` argument is not passed, this parameter is bound to the default list object, which is initially empty.

❷ This assignment makes `self.passengers` an alias for `passengers`, which is itself an alias for the default list, when no `passengers` argument is given.

❸ When the methods `.remove()` and `.append()` are used with `self.passengers`, we are actually mutating the default list, which is an attribute of the function object.

Example 6-13 shows the eerie behavior of the `HauntedBus`.

Example 6-13. Buses haunted by ghost passengers

```
>>> bus1 = HauntedBus(['Alice', 'Bill'])  ❶
>>> bus1.passengers
['Alice', 'Bill']
>>> bus1.pick('Charlie')
>>> bus1.drop('Alice')
>>> bus1.passengers  ❷
['Bill', 'Charlie']
>>> bus2 = HauntedBus()  ❸
>>> bus2.pick('Carrie')
>>> bus2.passengers
['Carrie']
>>> bus3 = HauntedBus()  ❹
>>> bus3.passengers  ❺
['Carrie']
>>> bus3.pick('Dave')
>>> bus2.passengers  ❻
['Carrie', 'Dave']
>>> bus2.passengers is bus3.passengers  ❼
True
>>> bus1.passengers  ❽
['Bill', 'Charlie']
```

❶ `bus1` starts with a two-passenger list.

❷ So far, so good: no surprises with `bus1`.

❸ `bus2` starts empty, so the default empty list is assigned to `self.passengers`.

❹ `bus3` also starts empty, again the default list is assigned.

❺ The default is no longer empty!

❻ Now `Dave`, picked by `bus3`, appears in `bus2`.

❼ The problem: `bus2.passengers` and `bus3.passengers` refer to the same list.

❽ But `bus1.passengers` is a distinct list.

The problem is that `HauntedBus` instances that don't get an initial passenger list end up sharing the same passenger list among themselves.

Such bugs may be subtle. As Example 6-13 demonstrates, when a `HauntedBus` is instantiated with passengers, it works as expected. Strange things happen only when a `HauntedBus` starts empty, because then `self.passengers` becomes an alias for the default value of the `passengers` parameter. The problem is that each default value is evaluated when the function is defined—i.e., usually when the module is loaded—and the default values become attributes of the function object. So if a default value is a mutable object, and you change it, the change will affect every future call of the function.

After running the lines in Example 6-13, you can inspect the `HauntedBus.__init__` object and see the ghost students haunting its `__defaults__` attribute:

```
>>> dir(HauntedBus.__init__)  # doctest: +ELLIPSIS
['__annotations__', '__call__', ..., '__defaults__', ...]
>>> HauntedBus.__init__.__defaults__
(['Carrie', 'Dave'],)
```

Finally, we can verify that `bus2.passengers` is an alias bound to the first element of the `HauntedBus.__init__.__defaults__` attribute:

```
>>> HauntedBus.__init__.__defaults__[0] is bus2.passengers
True
```

The issue with mutable defaults explains why `None` is commonly used as the default value for parameters that may receive mutable values. In Example 6-8, `__init__` checks whether the `passengers` argument is `None`. If it is, `self.passengers` is bound to a new empty list. If `passengers` is not `None`, the correct implementation binds a copy of that argument to `self.passengers`. The next section explains why copying the argument is a good practice.

Defensive Programming with Mutable Parameters

When you are coding a function that receives a mutable parameter, you should carefully consider whether the caller expects the argument passed to be changed.

For example, if your function receives a `dict` and needs to modify it while processing it, should this side effect be visible outside of the function or not? Actually it depends on the context. It's really a matter of aligning the expectation of the coder of the function and that of the caller.

The last bus example in this chapter shows how a `TwilightBus` breaks expectations by sharing its passenger list with its clients. Before studying the implementation, see in Example 6-14 how the `TwilightBus` class works from the perspective of a client of the class.

Example 6-14. Passengers disappear when dropped by a `TwilightBus`

```
>>> basketball_team = ['Sue', 'Tina', 'Maya', 'Diana', 'Pat']  ❶
>>> bus = TwilightBus(basketball_team)  ❷
>>> bus.drop('Tina')  ❸
>>> bus.drop('Pat')
>>> basketball_team  ❹
['Sue', 'Maya', 'Diana']
```

❶ `basketball_team` holds five student names.

❷ A `TwilightBus` is loaded with the team.

❸ The `bus` drops one student, then another.

❹ The dropped passengers vanished from the basketball team!

`TwilightBus` violates the "Principle of least astonishment," a best practice of interface design.[3] It surely is astonishing that when the bus drops a student, their name is removed from the basketball team roster.

Example 6-15 is the implementation `TwilightBus` and an explanation of the problem.

Example 6-15. A simple class to show the perils of mutating received arguments

```
class TwilightBus:
    """A bus model that makes passengers vanish"""

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

    def pick(self, name):
        self.passengers.append(name)

    def drop(self, name):
        self.passengers.remove(name)  ❸
```

3 See *Principle of least astonishment* in the English Wikipedia.

❶ Here we are careful to create a new empty list when `passengers` is `None`.

❷ However, this assignment makes `self.passengers` an alias for `passengers`, which is itself an alias for the actual argument passed to `__init__` (i.e., `basketball_team` in Example 6-14).

❸ When the methods `.remove()` and `.append()` are used with `self.passengers`, we are actually mutating the original list received as an argument to the constructor.

The problem here is that the bus is aliasing the list that is passed to the constructor. Instead, it should keep its own passenger list. The fix is simple: in `__init__`, when the `passengers` parameter is provided, `self.passengers` should be initialized with a copy of it, as we did correctly in Example 6-8:

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

❶ Make a copy of the `passengers` list, or convert it to a `list` if it's not one.

Now our internal handling of the passenger list will not affect the argument used to initialize the bus. As a bonus, this solution is more flexible: now the argument passed to the `passengers` parameter may be a `tuple` or any other iterable, like a `set` or even database results, because the `list` constructor accepts any iterable. As we create our own list to manage, we ensure that it supports the necessary `.remove()` and `.append()` operations we use in the `.pick()` and `.drop()` methods.

Unless a method is explicitly intended to mutate an object received as an argument, you should think twice before aliasing the argument object by simply assigning it to an instance variable in your class. If in doubt, make a copy. Your clients will be happier. Of course, making a copy is not free: there is a cost in CPU and memory. However, an API that causes subtle bugs is usually a bigger problem than one that is a little slower or uses more resources.

Now let's talk about one of the most misunderstood of Python's statements: `del`.

del and Garbage Collection

> Objects are never explicitly destroyed; however, when they become unreachable they may be garbage-collected.
>
> —"Data Model" chapter of *The Python Language Reference*

The first strange fact about `del` is that it's not a function, it's a statement. We write `del x` and not `del(x)`—although the latter also works, but only because the expressions `x` and `(x)` usually mean the same thing in Python.

The second surprising fact is that `del` deletes references, not objects. Python's garbage collector may discard an object from memory as an indirect result of `del`, if the deleted variable was the last reference to the object. Rebinding a variable may also cause the number of references to an object to reach zero, causing its destruction.

```
>>> a = [1, 2]  ❶
>>> b = a       ❷
>>> del a       ❸
>>> b           ❹
[1, 2]
>>> b = [3]     ❺
```

❶ Create object `[1, 2]` and bind `a` to it.

❷ Bind `b` to the same `[1, 2]` object.

❸ Delete reference `a`.

❹ `[1, 2]` was not affected, because `b` still points to it.

❺ Rebinding `b` to a different object removes the last remaining reference to `[1, 2]`. Now the garbage collector can discard that object.

There is a `__del__` special method, but it does not cause the disposal of the instance, and should not be called by your code. `__del__` is invoked by the Python interpreter when the instance is about to be destroyed to give it a chance to release external resources. You will seldom need to implement `__del__` in your own code, yet some Python programmers spend time coding it for no good reason. The proper use of `__del__` is rather tricky. See the `__del__` special method documentation in the "Data Model" chapter of *The Python Language Reference*.

In CPython, the primary algorithm for garbage collection is reference counting. Essentially, each object keeps count of how many references point to it. As soon as

that *refcount* reaches zero, the object is immediately destroyed: CPython calls the `__del__` method on the object (if defined) and then frees the memory allocated to the object. In CPython 2.0, a generational garbage collection algorithm was added to detect groups of objects involved in reference cycles—which may be unreachable even with outstanding references to them, when all the mutual references are contained within the group. Other implementations of Python have more sophisticated garbage collectors that do not rely on reference counting, which means the `__del__` method may not be called immediately when there are no more references to the object. See "PyPy, Garbage Collection, and a Deadlock" by A. Jesse Jiryu Davis for discussion of improper and proper use of `__del__`.

To demonstrate the end of an object's life, Example 6-16 uses `weakref.finalize` to register a callback function to be called when an object is destroyed.

Example 6-16. Watching the end of an object when no more references point to it

```
>>> import weakref
>>> s1 = {1, 2, 3}
>>> s2 = s1                ❶
>>> def bye():             ❷
...     print('...like tears in the rain.')
...
>>> ender = weakref.finalize(s1, bye)  ❸
>>> ender.alive  ❹
True
>>> del s1
>>> ender.alive  ❺
True
>>> s2 = 'spam'  ❻
...like tears in the rain.
>>> ender.alive
False
```

❶ `s1` and `s2` are aliases referring to the same set, `{1, 2, 3}`.

❷ This function must not be a bound method of the object about to be destroyed or otherwise hold a reference to it.

❸ Register the `bye` callback on the object referred by `s1`.

❹ The `.alive` attribute is `True` before the `finalize` object is called.

❺ As discussed, `del` did not delete the object, just the `s1` reference to it.

❻ Rebinding the last reference, `s2`, makes `{1, 2, 3}` unreachable. It is destroyed, the `bye` callback is invoked, and `ender.alive` becomes `False`.

The point of Example 6-16 is to make explicit that `del` does not delete objects, but objects may be deleted as a consequence of being unreachable after `del` is used.

You may be wondering why the `{1, 2, 3}` object was destroyed in Example 6-16. After all, the `s1` reference was passed to the `finalize` function, which must have held on to it in order to monitor the object and invoke the callback. This works because `finalize` holds a *weak reference* to `{1, 2, 3}`. Weak references to an object do not increase its reference count. Therefore, a weak reference does not prevent the target object from being garbage collected. Weak references are useful in caching applications because you don't want the cached objects to be kept alive just because they are referenced by the cache.

Weak references is a very specialized topic. That's why I chose to skip it in this second edition. Instead, I published "Weak References" on *fluentpython.com*.

Tricks Python Plays with Immutables

This optional section discusses some Python details that are not really important for *users* of Python, and that may not apply to other Python implementations or even future versions of CPython. Nevertheless, I've seen people stumble upon these corner cases and then start using the `is` operator incorrectly, so I felt they were worth mentioning.

I was surprised to learn that, for a tuple `t`, `t[:]` does not make a copy, but returns a reference to the same object. You also get a reference to the same tuple if you write `tuple(t)`.[4] Example 6-17 proves it.

Example 6-17. A tuple built from another is actually the same exact tuple

```
>>> t1 = (1, 2, 3)
>>> t2 = tuple(t1)
>>> t2 is t1  ❶
True
>>> t3 = t1[:]
>>> t3 is t1  ❷
True
```

4 This is clearly documented. Type `help(tuple)` in the Python console to read: "If the argument is a tuple, the return value is the same object." I thought I knew everything about tuples before writing this book.

❶ `t1` and `t2` are bound to the same object.

❷ And so is `t3`.

The same behavior can be observed with instances of `str`, `bytes`, and `frozenset`. Note that a `frozenset` is not a sequence, so `fs[:]` does not work if `fs` is a `frozenset`. But `fs.copy()` has the same effect: it cheats and returns a reference to the same object, and not a copy at all, as Example 6-18 shows.[5]

Example 6-18. String literals may create shared objects

```
>>> t1 = (1, 2, 3)
>>> t3 = (1, 2, 3)  ❶
>>> t3 is t1  ❷
False
>>> s1 = 'ABC'
>>> s2 = 'ABC'  ❸
>>> s2 is s1 ❹
True
```

❶ Creating a new tuple from scratch.

❷ `t1` and `t3` are equal, but not the same object.

❸ Creating a second `str` from scratch.

❹ Surprise: `a` and `b` refer to the same `str`!

The sharing of string literals is an optimization technique called *interning*. CPython uses a similar technique with small integers to avoid unnecessary duplication of numbers that appear frequently in programs like 0, 1, –1, etc. Note that CPython does not intern all strings or integers, and the criteria it uses to do so is an undocumented implementation detail.

Never depend on `str` or `int` interning! Always use `==` instead of `is` to compare strings or integers for equality. Interning is an optimization for internal use of the Python interpreter.

5 The harmless lie of having the `copy` method not copying anything is justified by interface compatibility: it makes `frozenset` more compatible with `set`. Anyway, it makes no difference to the end user whether two identical immutable objects are the same or are copies.

The tricks discussed in this section, including the behavior of `frozenset.copy()`, are harmless "lies" that save memory and make the interpreter faster. Do not worry about them, they should not give you any trouble because they only apply to immutable types. Probably the best use of these bits of trivia is to win bets with fellow Pythonistas.[6]

Chapter Summary

Every Python object has an identity, a type, and a value. Only the value of an object may change over time.[7]

If two variables refer to immutable objects that have equal values (`a == b` is `True`), in practice it rarely matters if they refer to copies or are aliases referring to the same object, because the value of an immutable object does not change, with one exception. The exception being immutable collections such as tuples: if an immutable collection holds references to mutable items, then its value may actually change when the value of a mutable item changes. In practice, this scenario is not so common. What never changes in an immutable collection are the identities of the objects within. The `frozenset` class does not suffer from this problem because it can only hold hashable elements, and the value of hashable objects cannot ever change, by definition.

The fact that variables hold references has many practical consequences in Python programming:

- Simple assignment does not create copies.
- Augmented assignment with `+=` or `*=` creates new objects if the lefthand variable is bound to an immutable object, but may modify a mutable object in place.
- Assigning a new value to an existing variable does not change the object previously bound to it. This is called a rebinding: the variable is now bound to a different object. If that variable was the last reference to the previous object, that object will be garbage collected.

6 A terrible use for this information would be to ask about it when interviewing candidates or authoring questions for "certification" exams. There are countless more important and useful facts to check for Python knowledge.

7 Actually the type of an object may be changed by merely assigning a different class to its `__class__` attribute, but that is pure evil and I regret writing this footnote.

- Function parameters are passed as aliases, which means the function may change any mutable object received as an argument. There is no way to prevent this, except making local copies or using immutable objects (e.g., passing a tuple instead of a list).
- Using mutable objects as default values for function parameters is dangerous because if the parameters are changed in place, then the default is changed, affecting every future call that relies on the default.

In CPython, objects are discarded as soon as the number of references to them reaches zero. They may also be discarded if they form groups with cyclic references but not outside references.

In some situations, it may be useful to hold a reference to an object that will not—by itself—keep an object alive. One example is a class that wants to keep track of all its current instances. This can be done with weak references, a low-level mechanism underlying the more useful collections `WeakValueDictionary`, `WeakKeyDictionary`, `WeakSet`, and the `finalize` function from the `weakref` module. For more on this, please see "Weak References" at *fluentpython.com*.

Further Reading

The "Data Model" chapter of *The Python Language Reference* starts with a clear explanation of object identities and values.

Wesley Chun, author of the *Core Python* series of books, presented Understanding Python's Memory Model, Mutability, and Methods at EuroPython 2011, covering not only the theme of this chapter but also the use of special methods.

Doug Hellmann wrote the posts "copy – Duplicate Objects" and "weakref—Garbage-Collectable References to Objects" covering some of the topics we just discussed.

More information on the CPython generational garbage collector can be found in the gc module documentation, which starts with the sentence "This module provides an interface to the optional garbage collector." The "optional" qualifier here may be surprising, but the "Data Model" chapter also states:

> An implementation is allowed to postpone garbage collection or omit it altogether—it is a matter of implementation quality how garbage collection is implemented, as long as no objects are collected that are still reachable.

Pablo Galindo wrote more in-depth treatment of Python's GC in "Design of CPython's Garbage Collector" in the *Python Developer's Guide*, aimed at new and experienced contributors to the CPython implementation.

The CPython 3.4 garbage collector improved handling of objects with a `__del__` method, as described in PEP 442—Safe object finalization.

Wikipedia has an article about string interning, mentioning the use of this technique in several languages, including Python.

Wikipedia also as an article on "Haddocks' Eyes", the Lewis Carroll song I quoted at the top of this chapter. The Wikipedia editors wrote that the lyrics are used in works on logic and philosophy "to elaborate on the symbolic status of the concept of *name*: a name as identification marker may be assigned to anything, including another name, thus introducing different levels of symbolization."

Soapbox

Equal Treatment to All Objects

I learned Java before I discovered Python. The == operator in Java never felt right to me. It is much more common for programmers to care about equality than identity, but for objects (not primitive types), the Java == compares references, and not object values. Even for something as basic as comparing strings, Java forces you to use the `.equals` method. Even then, there is another catch: if you write `a.equals(b)` and `a` is `null`, you get a null pointer exception. The Java designers felt the need to overload + for strings, so why not go ahead and overload == as well?

Python gets this right. The == operator compares object values; `is` compares references. And because Python has operator overloading, == works sensibly with all objects in the standard library, including `None`, which is a proper object, unlike Java's `null`.

And of course, you can define `__eq__` in your own classes to decide what == means for your instances. If you don't override `__eq__`, the method inherited from `object` compares object IDs, so the fallback is that every instance of a user-defined class is considered different.

These are some of the things that made me switch from Java to Python as soon as I finished reading *The Python Tutorial* one afternoon in September 1998.

Mutability

This chapter would not be necessary if all Python objects were immutable. When you are dealing with unchanging objects, it makes no difference whether variables hold the actual objects or references to shared objects. If `a == b` is true, and neither object

can change, they might as well be the same. That's why string interning is safe. Object identity becomes important only when objects are mutable.

In "pure" functional programming, all data is immutable: appending to a collection actually creates a new collection. Elixir is one easy to learn, practical functional language in which all built-in types are immutable, including lists.

Python, however, is not a functional language, much less a pure one. Instances of user-defined classes are mutable by default in Python—as in most object-oriented languages. When creating your own objects, you have to be extra careful to make them immutable, if that is a requirement. Every attribute of the object must also be immutable, otherwise you end up with something like the `tuple`: immutable as far as object IDs go, but the value of a `tuple` may change if it holds a mutable object.

Mutable objects are also the main reason why programming with threads is so hard to get right: threads mutating objects without proper synchronization produce corrupted data. Excessive synchronization, on the other hand, causes deadlocks. The Erlang language and platform—which includes Elixir—was designed to maximize uptime in highly concurrent, distributed applications such as telecommunications switches. Naturally, they chose immutable data by default.

Object Destruction and Garbage Collection

There is no mechanism in Python to directly destroy an object, and this omission is actually a great feature: if you could destroy an object at any time, what would happen to existing references pointing to it?

Garbage collection in CPython is done primarily by reference counting, which is easy to implement, but is prone to memory leaking when there are reference cycles, so with version 2.0 (October 2000) a generational garbage collector was implemented, and it is able to dispose of unreachable objects kept alive by reference cycles.

But the reference counting is still there as a baseline, and it causes the immediate disposal of objects with zero references. This means that, in CPython—at least for now—it's safe to write this:

```
open('test.txt', 'wt', encoding='utf-8').write('1, 2, 3')
```

That code is safe because the reference count of the file object will be zero after the `write` method returns, and Python will immediately close the file before destroying the object representing it in memory. However, the same line is not safe in Jython or IronPython that use the garbage collector of their host runtimes (the Java VM and the .NET CLR), which are more sophisticated but do not rely on reference counting and may take longer to destroy the object and close the file. In all cases, including CPython, the best practice is to explicitly close the file, and the most reliable way of doing it is using the `with` statement, which guarantees that the file will be closed even if exceptions are raised while it is open. Using `with`, the previous snippet becomes:

```
with open('test.txt', 'wt', encoding='utf-8') as fp:
    fp.write('1, 2, 3')
```

If you are into the subject of garbage collectors, you may want to read Thomas Perl's paper "Python Garbage Collector Implementations: CPython, PyPy and GaS", from which I learned the bit about the safety of the `open().write()` in CPython.

Parameter Passing: Call by Sharing

A popular way of explaining how parameter passing works in Python is the phrase: "Parameters are passed by value, but the values are references." This is not wrong, but causes confusion because the most common parameter passing modes in older languages are *call by value* (the function gets a copy of the argument) and *call by reference* (the function gets a pointer to the argument). In Python, the function gets a copy of the arguments, but the arguments are always references. So the value of the referenced objects may be changed, if they are mutable, but their identity cannot. Also, because the function gets a copy of the reference in an argument, rebinding it in the function body has no effect outside of the function. I adopted the term *call by sharing* after reading up on the subject in *Programming Language Pragmatics*, 3rd ed., by Michael L. Scott (Morgan Kaufmann), section "8.3.1: Parameter Modes."

PART II

Functions as Objects