`del` and garbage collection

> Objects are never explicitly destroyed; however, when they become unreachable they may be garbage-collected.
>
> — Python Language Reference: Data model

The `del` statement deletes names, not objects. An object may be garbage collected as result of a `del` command, but only if the variable deleted holds the last reference to the object, or if the object becomes unreachable[4]. Rebinding a variable may also cause the number of references to an object reach zero, causing its destruction.

4. If two objects refer to each other, as in Example 8-10, they may be destroyed if the garbage collector determines that they are otherwise unreachable because their only references are their mutual references.

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 beginners spend time coding it for no good reason. The proper use of `__del__` is rather tricky. See the `__del__` special method documentation in the Python data model chapter of the 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 collector that do not rely of reference counting, which means the `__del__` method may not be called immediately when there are no more references to the object. See the PyPy, Garbage Collection, And A Deadlock by A. Jesse Jiryu Davis for discussion of improper and proper use of `__del__`.

In order to demonstrate the end of an object's life, Example 8-16 uses `weakref.finalize` to register a callback function to be called when an object is destroyed.

Example 8-16. Watching the end of an object when no more references point to it.

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

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

❷ This function must not be a bound method 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` does not delete an object, just a 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 8-16 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 8-16. 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}`, as explained in the next section.

Weak references

The presence of references is what keeps an object alive in memory. When the reference count of an object reaches zero, the garbage collector disposes of it. But sometimes it is useful to have a reference to an object that does not keep it around longer than necessary. A common use case is a cache.

Weak references to an object do not increase its reference count. The object that is the target of a reference is called the *referent*. Therefore, we say that a weak reference does not prevent the referent 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.

Example 8-17 shows how a `weakref.ref` instance can be called to reach its referent. If the object is alive, calling the weak reference returns it, otherwise `None` is returned.

Example 8-17 is a console session, and the Python console automatically binds the `_` variable to the result of expressions that are not `None`. This interfered with my intended demonstration but also highlights a practical matter: when trying to micro-manage memory we are often surprised by hidden, implicit assignments that create new references to our objects. The `_` console variable is one example. Traceback objects are another common source of unexpected references.

Example 8-17. A weak reference is a callable that returns the referenced object or `None` if the referent is no more.

```
>>> import weakref
>>> a_set = {0, 1}
>>> wref = weakref.ref(a_set)
>>> wref
<weakref at 0x100637598; to 'set' at 0x100636748>
>>> wref()  ❷
{0, 1}
>>> a_set = {2, 3, 4}  ❸
>>> wref()  ❹
{0, 1}
>>> wref() is None  ❺
False
>>> wref() is None  ❻
True
```

The `wref` weak reference object is created and inspected in the next line.

❷ Invoking `wref()` returns the referenced object, `{0, 1}`. Because this is a console session, the result `{0, 1}` is bound to the _ variable.

❸ `a_set` no longer refers to the `{0, 1}` set, so its reference count is decreased. But the _ variable still refers to it.

❹ Calling `wref()` still returns `{0, 1}`.

❺ When this expression is evaluated, `{0, 1}` lives, therefore `wref()` is not `None`. But _ is then bound to the resulting value, `False`. Now there are no more strong references to `{0, 1}`.

❻ Because the `{0, 1}` object is now gone, this last call to `wref()` returns `None`.

The `weakref` module documentation makes the point that the `weakref.ref` class is actually a low-level interface intended for advanced uses, and that most programs are better served by the use of the `weakref` collections and `finalize`. In other words, consider using `WeakKeyDictionary`, `WeakValueDictionary`, `WeakSet` and `finalize` — which use weak references internally — instead of creating and handling your own `weakref.ref` instances by hand. We just did that in Example 8-17 in the hope that showing a single `weakref.ref` in action could take away some of the mistery around them. But in practice, most of the time Python programs use the `weakref` collections.

The next subsection briefly discusses the `weakref` collections.

The `WeakValueDictionary` skit

The class `WeakValueDictionary` implements a mutable mapping where the values are weak references to objects. When a referred object is garbage collected elsewhere in the

program, the corresponding key is automatically removed from the `WeakValueDictionary`. This is commonly used for caching.

Our demonstration of a `WeakValueDictionary` is inspired by the classic *Cheese Shop* skit by Monty Python, in which a customer asks for more than 40 kinds of cheese, including cheddar and mozzarella, but none are in stock[5].

Example 8-18 implements a trivial class to represent each kind of cheese.

Example 8-18. `Cheese` has a `kind` attribute and a standard representation.

```
class Cheese:

    def __init__(self, kind):
        self.kind = kind

    def __repr__(self):
        return 'Cheese(%r)' % self.kind
```

In Example 8-19 each cheese is loaded from a `catalog` to a `stock` implemented as a `WeakValueDictionary`. However, all but one disappear from the `stock` as soon as the `catalog` is deleted. Can you explain why the Parmesan cheese lasts longer than the others[6]? The tip after the code has the answer.

Example 8-19. Customer: "Have you in fact got any cheese here at all?"

```
>>> import weakref
>>> stock = weakref.WeakValueDictionary()
>>> catalog = [Cheese('Red Leicester'), Cheese('Tilsit'),
...                Cheese('Brie'), Cheese('Parmesan')]
...
>>> for cheese in catalog:
...     stock[cheese.kind] = cheese  ❷
...
>>> sorted(stock.keys())
['Brie', 'Parmesan', 'Red Leicester', 'Tilsit']  ❸
>>> del catalog
>>> sorted(stock.keys())
['Parmesan']  ❹
>>> del cheese
>>> sorted(stock.keys())
[]
```

5. `cheeseshop.python.org` is also an alias for PyPI — the Python Package Index software repository — which started its life quite empty. At this writing the Python Cheese Shop has 41,426 packages. Not bad, but still far from the more than 131,000 modules available in CPAN — the Comprehensive Perl Archive Network — the envy of all dynamic language communities.

6. Parmesan cheese is aged at least a year at the factory, so it is more durable than fresh cheese, but this is not the answer we are looking for.

`stock` is a `WeakValueDictionary`

❷ The `stock` maps the name of the cheese to a weak reference to the cheese instance in the `catalog`.

❸ The `stock` is complete.

❹ After the `catalog` is deleted, most cheeses are gone from the `stock`, as expected in `WeakValueDictionary`. Why not all, in this case?

A temporary variable may cause an object to last longer than expected by holding a reference to it. This is usually not a problem with local variables: they are destroyed when the function returns. But in Example 8-19, the `for` loop variable `cheese` is a global variable and will never go away unless explicitly deleted.

A counterpart to the `WeakValueDictionary` is the `WeakKeyDictionary` in which the keys are weak references. The `weakref.WeakKeyDictionary` documentation hints on possible uses:

> [A `WeakKeyDictionary`] can be used to associate additional data with an object owned by other parts of an application without adding attributes to those objects. This can be especially useful with objects that override attribute accesses.

The `weakref` module also provides a `WeakSet`, simply described in the docs as "Set class that keeps weak references to its elements. An element will be discarded when no strong reference to it exists any more." If you need to build a class that is aware of every one of its instances, a good solution is to create a class attribute with a `WeakSet` to hold the references to the instances. Otherwise, if a regular `set` was used, the instances would never be garbage collected, because the class itself would have strong references to them, and classes live as long as the Python process unless you deliberately delete them.

These collections, and weak references in general are limited in the kinds of objects they can handle. The next section explains.

Limitations of weak references

Not every Python object may be the target, or referent, of a weak reference. Basic `list` and `dict` instances may not be referents, but a plain subclass of either can solve this problem easily:

```
class MyList(list):
    """list subclass whose instances may be weakly referenced"""

a_list = MyList(range(10))
```

```
# a_list can be the target of a weak reference
wref_to_a_list = weakref.ref(a_list)
```

A `set` instance can be a referent, that's why a `set` was used in Example 8-17. User-defined types also pose no problem, which explains why the silly `Cheese` class was needed in Example 8-19. But `int` and `tuple` instances cannot be targets of weak references, even if subclasses of those types are created.

Most of these limitations are implementation details of CPython that may not apply to other Python iterpreters. They are the result of internal optimizations, some of which are discussed in the following (highly optional) section.

Tricks Python plays with immutables

You may safely skip this section. It discusses some Python implementation details that are not really important for *users* of Python. They are shortcuts and optimizations done by the CPython core developers which should not bother you when using the language, and that may not apply to other Python implementations or even future versions of CPython. Nevertheless, while experimenting with aliases and copies you may stumble upon these tricks, 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)`. [7]

Example 8-20. A tuple built from another is actually the same exact tuple.

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

`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

7. 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.