

## 8.8. `weakref` — Weak references

**Source code:** [Lib/weakref.py](#)

The `weakref` module allows the Python programmer to create *weak references* to objects.

In the following, the term *referent* means the object which is referred to by a weak reference.

A weak reference to an object is not enough to keep the object alive: when the only remaining references to a referent are weak references, [garbage collection](#) is free to destroy the referent and reuse its memory for something else. However, until the object is actually destroyed the weak reference may return the object even if there are no strong references to it.

A primary use for weak references is to implement caches or mappings holding large objects, where it's desired that a large object not be kept alive solely because it appears in a cache or mapping.

For example, if you have a number of large binary image objects, you may wish to associate a name with each. If you used a Python dictionary to map names to images, or images to names, the image objects would remain alive just because they appeared as values or keys in the dictionaries. The `WeakKeyDictionary` and `WeakValueDictionary` classes supplied by the `weakref` module are an alternative, using weak references to construct mappings that don't keep objects alive solely because they appear in the mapping objects. If, for example, an image object is a value in a `WeakValueDictionary`, then when the last remaining references to that image object are the weak references held by weak mappings, garbage collection can reclaim the object, and its corresponding entries in weak mappings are simply deleted.

`WeakKeyDictionary` and `WeakValueDictionary` use weak references in their implementation, setting up callback functions on the weak references that notify the weak dictionaries when a key or value has been reclaimed by garbage collection. `WeakSet` implements the `set` interface, but keeps weak references to its elements, just like a `WeakKeyDictionary` does.

`finalize` provides a straight forward way to register a cleanup function to be called when an object is garbage collected. This is simpler to use than setting up a callback function on a raw weak reference, since the module automatically ensures that the finalizer remains alive until the object is collected.

Most programs should find that using one of these weak container types or [finalize](#) is all they need – it's not usually necessary to create your own weak references directly. The low-level machinery is exposed by the [weakref](#) module for the benefit of advanced uses.

Not all objects can be weakly referenced; those objects which can include class instances, functions written in Python (but not in C), instance methods, sets, frozensets, some [file objects](#), [generators](#), type objects, sockets, arrays, dequeues, regular expression pattern objects, and code objects.

*Changed in version 3.2:* Added support for `thread.lock`, `threading.Lock`, and code objects.

Several built-in types such as [list](#) and [dict](#) do not directly support weak references but can add support through subclassing:

```
class Dict(dict):
    pass

obj = Dict(red=1, green=2, blue=3)  # this object is weak referenced
```

Other built-in types such as [tuple](#) and [int](#) do not support weak references even when subclassed (This is an implementation detail and may be different across various Python implementations.).

Extension types can easily be made to support weak references; see [Weak Reference Support](#).

`class weakref.ref(object[, callback])`

Return a weak reference to *object*. The original object can be retrieved by calling the reference object if the referent is still alive; if the referent is no longer alive, calling the reference object will cause [None](#) to be returned. If *callback* is provided and not [None](#), and the returned weakref object is still alive, the callback will be called when the object is about to be finalized; the weak reference object will be passed as the only parameter to the callback; the referent will no longer be available.

It is allowable for many weak references to be constructed for the same object. Callbacks registered for each weak reference will be called from the most recently registered callback to the oldest registered callback.

Exceptions raised by the callback will be noted on the standard error output, but cannot be propagated; they are handled in exactly the same way as exceptions raised from an object's `__del__()` method.

Weak references are [hashable](#) if the *object* is hashable. They will maintain their hash value even after the *object* was deleted. If `hash()` is called the first time only after the *object* was deleted, the call will raise [TypeError](#).

Weak references support tests for equality, but not ordering. If the referents are still alive, two references have the same equality relationship as their referents (regardless of the *callback*). If either referent has been deleted, the references are equal only if the reference objects are the same object.

This is a subclassable type rather than a factory function.

### **`__callback__`**

This read-only attribute returns the callback currently associated to the weakref. If there is no callback or if the referent of the weakref is no longer alive then this attribute will have value `None`.

*Changed in version 3.4:* Added the `__callback__` attribute.

`weakref.proxy(object[, callback])`

Return a proxy to *object* which uses a weak reference. This supports use of the proxy in most contexts instead of requiring the explicit dereferencing used with weak reference objects. The returned object will have a type of either `ProxyType` or `CallableProxyType`, depending on whether *object* is callable. Proxy objects are not [hashable](#) regardless of the referent; this avoids a number of problems related to their fundamentally mutable nature, and prevent their use as dictionary keys. *callback* is the same as the parameter of the same name to the `ref()` function.

`weakref.getweakrefcount(object)`

Return the number of weak references and proxies which refer to *object*.

`weakref.getweakrefs(object)`

Return a list of all weak reference and proxy objects which refer to *object*.

`class weakref.WeakKeyDictionary([dict])`

Mapping class that references keys weakly. Entries in the dictionary will be discarded when there is no longer a strong reference to the key. This 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.

**Note:** Caution: Because a [WeakKeyDictionary](#) is built on top of a Python dictionary, it must not change size when iterating over it. This can be difficult to ensure for a [WeakKeyDictionary](#) because actions performed by the pro-

gram during iteration may cause items in the dictionary to vanish “by magic” (as a side effect of garbage collection).

`WeakKeyDictionary` objects have an additional method that exposes the internal references directly. The references are not guaranteed to be “live” at the time they are used, so the result of calling the references needs to be checked before being used. This can be used to avoid creating references that will cause the garbage collector to keep the keys around longer than needed.

`WeakKeyDictionary.keyrefs()`

Return an iterable of the weak references to the keys.

`class weakref.WeakValueDictionary([dict])`

Mapping class that references values weakly. Entries in the dictionary will be discarded when no strong reference to the value exists any more.

**Note:** Caution: Because a `WeakValueDictionary` is built on top of a Python dictionary, it must not change size when iterating over it. This can be difficult to ensure for a `WeakValueDictionary` because actions performed by the program during iteration may cause items in the dictionary to vanish “by magic” (as a side effect of garbage collection).

`WeakValueDictionary` objects have an additional method that has the same issues as the `keyrefs()` method of `WeakKeyDictionary` objects.

`WeakValueDictionary.valuerefs()`

Return an iterable of the weak references to the values.

`class weakref.WeakSet([elements])`

Set class that keeps weak references to its elements. An element will be discarded when no strong reference to it exists any more.

`class weakref.WeakMethod(method)`

A custom `ref` subclass which simulates a weak reference to a bound method (i.e., a method defined on a class and looked up on an instance). Since a bound method is ephemeral, a standard weak reference cannot keep hold of it. `WeakMethod` has special code to recreate the bound method until either the object or the original function dies:

```
>>> class C:
...     def method(self):
...         print("method called!")
...
>>> c = C()
```

```
>>>
```

```

>>> r = weakref.ref(c.method)
>>> r()
>>> r = weakref.WeakMethod(c.method)
>>> r()
<bound method C.method of <__main__.C object at 0x7fc859830220>>
>>> r()()
method called!
>>> del c
>>> gc.collect()
0
>>> r()
>>>

```

*New in version 3.4.*

`class weakref.finalize(obj, func, *args, **kwargs)`

Return a callable finalizer object which will be called when *obj* is garbage collected. Unlike an ordinary weak reference, a finalizer will always survive until the reference object is collected, greatly simplifying lifecycle management.

A finalizer is considered *alive* until it is called (either explicitly or at garbage collection), and after that it is *dead*. Calling a live finalizer returns the result of evaluating `func(*arg, **kwargs)`, whereas calling a dead finalizer returns [None](#).

Exceptions raised by finalizer callbacks during garbage collection will be shown on the standard error output, but cannot be propagated. They are handled in the same way as exceptions raised from an object's `__del__()` method or a weak reference's callback.

When the program exits, each remaining live finalizer is called unless its [atexit](#) attribute has been set to false. They are called in reverse order of creation.

A finalizer will never invoke its callback during the later part of the [interpreter shutdown](#) when module globals are liable to have been replaced by [None](#).

**`__call__()`**

If *self* is alive then mark it as dead and return the result of calling `func(*args, **kwargs)`. If *self* is dead then return [None](#).

**`detach()`**

If *self* is alive then mark it as dead and return the tuple (obj, func, args, kwargs). If *self* is dead then return [None](#).

**`peek()`**

If *self* is alive then return the tuple (obj, func, args, kwargs). If *self* is dead then return [None](#).

## **alive**

Property which is true if the finalizer is alive, false otherwise.

## **atexit**

A writable boolean property which by default is true. When the program exits, it calls all remaining live finalizers for which [atexit](#) is true. They are called in reverse order of creation.

**Note:** It is important to ensure that *func*, *args* and *kwargs* do not own any references to *obj*, either directly or indirectly, since otherwise *obj* will never be garbage collected. In particular, *func* should not be a bound method of *obj*.

*New in version 3.4.*

### `weakref.ReferenceType`

The type object for weak references objects.

### `weakref.ProxyType`

The type object for proxies of objects which are not callable.

### `weakref.CallableProxyType`

The type object for proxies of callable objects.

### `weakref.ProxyTypes`

Sequence containing all the type objects for proxies. This can make it simpler to test if an object is a proxy without being dependent on naming both proxy types.

### *exception* `weakref.ReferenceError`

Exception raised when a proxy object is used but the underlying object has been collected. This is the same as the standard [ReferenceError](#) exception.

#### **See also:**

#### **[PEP 205](#) - Weak References**

The proposal and rationale for this feature, including links to earlier implementations and information about similar features in other languages.

## 8.8.1. Weak Reference Objects

Weak reference objects have no methods and no attributes besides [ref.\\_\\_callback\\_\\_](#). A weak reference object allows the referent to be obtained, if it still exists, by calling it:

```
>>> import weakref
>>> class Object:
...     pass
...
>>> o = Object()
>>> r = weakref.ref(o)
>>> o2 = r()
>>> o is o2
True
```

```
>>>
```

If the referent no longer exists, calling the reference object returns `None`:

```
>>> del o, o2
>>> print(r())
None
```

```
>>>
```

Testing that a weak reference object is still live should be done using the expression `ref() is not None`. Normally, application code that needs to use a reference object should follow this pattern:

```
# r is a weak reference object
o = r()
if o is None:
    # referent has been garbage collected
    print("Object has been deallocated; can't frobnicate.")
else:
    print("Object is still live!")
    o.do_something_useful()
```

Using a separate test for “liveness” creates race conditions in threaded applications; another thread can cause a weak reference to become invalidated before the weak reference is called; the idiom shown above is safe in threaded applications as well as single-threaded applications.

Specialized versions of `ref` objects can be created through subclassing. This is used in the implementation of the `WeakValueDictionary` to reduce the memory overhead for each entry in the mapping. This may be most useful to associate additional information with a reference, but could also be used to insert additional processing on calls to retrieve the referent.

This example shows how a subclass of `ref` can be used to store additional information about an object and affect the value that’s returned when the referent is accessed:

```
import weakref

class ExtendedRef(weakref.ref):
```

```

def __init__(self, ob, callback=None, **annotations):
    super(ExtendedRef, self).__init__(ob, callback)
    self.__counter = 0
    for k, v in annotations.items():
        setattr(self, k, v)

def __call__(self):
    """Return a pair containing the referent and the number of
    times the reference has been called.
    """
    ob = super(ExtendedRef, self).__call__()
    if ob is not None:
        self.__counter += 1
        ob = (ob, self.__counter)
    return ob

```

## 8.8.2. Example

This simple example shows how an application can use object IDs to retrieve objects that it has seen before. The IDs of the objects can then be used in other data structures without forcing the objects to remain alive, but the objects can still be retrieved by ID if they do.

```

import weakref

_id2obj_dict = weakref.WeakValueDictionary()

def remember(obj):
    oid = id(obj)
    _id2obj_dict[oid] = obj
    return oid

def id2obj(oid):
    return _id2obj_dict[oid]

```

## 8.8.3. Finalizer Objects

The main benefit of using `finalize` is that it makes it simple to register a callback without needing to preserve the returned finalizer object. For instance

```

>>> import weakref
>>> class Object:
...     pass
...
>>> kenny = Object()
>>> weakref.finalize(kenny, print, "You killed Kenny!")
<finalize object at ...; for 'Object' at ...>

```



```
>>> del kenny
You killed Kenny!
```

The finalizer can be called directly as well. However the finalizer will invoke the callback at most once.

```
>>> def callback(x, y, z):
...     print("CALLBACK")
...     return x + y + z
...
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> assert f.alive
>>> assert f() == 6
CALLBACK
>>> assert not f.alive
>>> f()
# callback not called because finalizer de
>>> del obj
# callback not called because finalizer de
```

You can unregister a finalizer using its `detach()` method. This kills the finalizer and returns the arguments passed to the constructor when it was created.

```
>>> obj = Object()
>>> f = weakref.finalize(obj, callback, 1, 2, z=3)
>>> f.detach()
(<__main__.Object object ...>, <function callback ...>, (1, 2), {'z':
>>> newobj, func, args, kwargs = _
>>> assert not f.alive
>>> assert newobj is obj
>>> assert func(*args, **kwargs) == 6
CALLBACK
```

Unless you set the `atexit` attribute to `False`, a finalizer will be called when the program exits if it is still alive. For instance

```
>>> obj = Object()
>>> weakref.finalize(obj, print, "obj dead or exiting")
<finalize object at ...; for 'Object' at ...>
>>> exit()
obj dead or exiting
```

## 8.8.4. Comparing finalizers with `__del__()` methods

Suppose we want to create a class whose instances represent temporary directories. The directories should be deleted with their contents when the first of the following events occurs:

- the object is garbage collected,
- the object's `remove()` method is called, or
- the program exits.

We might try to implement the class using a `__del__()` method as follows:

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()

    def remove(self):
        if self.name is not None:
            shutil.rmtree(self.name)
            self.name = None

    @property
    def removed(self):
        return self.name is None

    def __del__(self):
        self.remove()
```

Starting with Python 3.4, `__del__()` methods no longer prevent reference cycles from being garbage collected, and module globals are no longer forced to `None` during [interpreter shutdown](#). So this code should work without any issues on CPython.

However, handling of `__del__()` methods is notoriously implementation specific, since it depends on internal details of the interpreter's garbage collector implementation.

A more robust alternative can be to define a finalizer which only references the specific functions and objects that it needs, rather than having access to the full state of the object:

```
class TempDir:
    def __init__(self):
        self.name = tempfile.mkdtemp()
        self._finalizer = weakref.finalize(self, shutil.rmtree, self.r

    def remove(self):
```

```
self._finalizer()

@property
def removed(self):
    return not self._finalizer.alive
```

Defined like this, our finalizer only receives a reference to the details it needs to clean up the directory appropriately. If the object never gets garbage collected the finalizer will still be called at exit.

The other advantage of weakref based finalizers is that they can be used to register finalizers for classes where the definition is controlled by a third party, such as running code when a module is unloaded:

```
import weakref, sys
def unloading_module():
    # implicit reference to the module globals from the function body
    weakref.finalize(sys.modules[__name__], unloading_module)
```

**Note:** If you create a finalizer object in a daemon thread just as the program exits then there is the possibility that the finalizer does not get called at exit. However, in a daemon thread `atexit.register()`, `try: ... finally: ...` and `with: ...` do not guarantee that cleanup occurs either.