

# weakref — Impermanent References to Objects

**Purpose:** Refer to an “expensive” object, but allow its memory to be reclaimed by the garbage collector if there are no other non-weak references.

The weakref module supports weak references to objects. A normal reference increments the reference count on the object and prevents it from being garbage collected. This outcome is not always desirable, especially when a circular reference might be present or when a cache of objects should be deleted when memory is needed. A weak reference is a handle to an object that does not keep it from being cleaned up automatically.

## References

Weak references to objects are managed through the ref class. To retrieve the original object, call the reference object.

```
# weakref_ref.py

import weakref

class ExpensiveObject:

    def __del__(self):
        print('(Deleting {})'.format(self))

obj = ExpensiveObject()
r = weakref.ref(obj)

print('obj:', obj)
print('ref:', r)
print('r():', r())

print('deleting obj')
del obj
print('r():', r())
```

In this case, since obj is deleted before the second call to the reference, the ref returns None.

```
$ python3 weakref_ref.py

obj: <__main__.ExpensiveObject object at 0x1007b1a58>
ref: <weakref at 0x1007a92c8; to 'ExpensiveObject' at 0x1007b1a58>
r(): <__main__.ExpensiveObject object at 0x1007b1a58>
deleting obj
(Deleting <__main__.ExpensiveObject object at 0x1007b1a58>)
r(): None
```

## Reference Callbacks

The ref constructor accepts an optional callback function that is invoked when the referenced object is deleted.

```
# weakref_ref_callback.py

import weakref

class ExpensiveObject:

    def __del__(self):
        print('(Deleting {})'.format(self))
```

```
def callback(reference):
    """Invoked when referenced object is deleted"""
    print('callback({!r})'.format(reference))

obj = ExpensiveObject()
r = weakref.ref(obj, callback)

print('obj:', obj)
print('ref:', r)
print('r():', r())

print('deleting obj')
del obj
print('r():', r())
```

The callback receives the reference object as an argument after the reference is “dead” and no longer refers to the original object. One use for this feature is to remove the weak reference object from a cache.

```
$ python3 weakref_ref_callback.py

obj: <__main__.ExpensiveObject object at 0x1010b1978>
ref: <weakref at 0x1010a92c8; to 'ExpensiveObject' at 0x1010b1978>
r(): <__main__.ExpensiveObject object at 0x1010b1978>
deleting obj
(Deleting <__main__.ExpensiveObject object at 0x1010b1978>)
callback(<weakref at 0x1010a92c8; dead>)
r(): None
```

## Finalizing Objects

For more robust management of resources when weak references are cleaned up, use `finalize` to associate callbacks with objects. A `finalize` instance is retained until the attached object is deleted, even if the application does not retain a reference to the finalizer.

```
# weakref_finalize.py

import weakref

class ExpensiveObject:
    def __del__(self):
        print('(Deleting {})'.format(self))

def on_finalize(*args):
    print('on_finalize({!r})'.format(args))

obj = ExpensiveObject()
weakref.finalize(obj, on_finalize, 'extra argument')

del obj
```

The arguments to `finalize` are the object to track, a callable to invoke when the object is garbage collected, and any positional or named arguments to pass to the callable.

```
$ python3 weakref_finalize.py

(Deleting <__main__.ExpensiveObject object at 0x1019b10f0>)
on_finalize(('extra argument',))
```

The `finalize` instance has a writable property `atexit` to control whether the callback is invoked as a program is exiting, if it hasn’t already been called.

```
# weakref_finalize_atexit.py
```

```
import sys
import weakref

class ExpensiveObject:

    def __del__(self):
        print('(Deleting {})'.format(self))

    def on_finalize(*args):
        print('on_finalize({!r})'.format(args))

obj = ExpensiveObject()
f = weakref.finalize(obj, on_finalize, 'extra argument')
f.atexit = bool(int(sys.argv[1]))
```

The default is to invoke the callback. Setting atexit to false disables that behavior.

```
$ python3 weakref_finalize_atexit.py 1

on_finalize(('extra argument',))
(Deleting <__main__.ExpensiveObject object at 0x1007b10f0>)

$ python3 weakref_finalize_atexit.py 0
```

Giving the finalize instance a reference to the object it tracks causes a reference to be retained, so the object is never garbage collected.

```
# weakref_finalize_reference.py
```

```
import gc
import weakref

class ExpensiveObject:

    def __del__(self):
        print('(Deleting {})'.format(self))

    def on_finalize(*args):
        print('on_finalize({!r})'.format(args))

obj = ExpensiveObject()
obj_id = id(obj)

f = weakref.finalize(obj, on_finalize, obj)
f.atexit = False

del obj

for o in gc.get_objects():
    if id(o) == obj_id:
        print('found uncollected object in gc')
```

As this example shows, even though the explicit reference to obj is deleted, the object is retained and visible to the garbage collector through f.

```
$ python3 weakref_finalize_reference.py

found uncollected object in gc
```

Using a bound method of a tracked object as the callable can also prevent an object from being finalized properly.

```
# weakref_finalize_reference_method.py

import gc
import weakref

class ExpensiveObject:

    def __del__(self):
        print('(Deleting {})'.format(self))

    def do_finalize(self):
        print('do_finalize')

obj = ExpensiveObject()
obj_id = id(obj)

f = weakref.finalize(obj, obj.do_finalize)
f.atexit = False

del obj

for o in gc.get_objects():
    if id(o) == obj_id:
        print('found uncollected object in gc')
```

Because the callable given to finalize is a bound method of the instance obj, the finalize object holds a reference to obj, which cannot be deleted and garbage collected.

```
$ python3 weakref_finalize_reference_method.py

found uncollected object in gc
```

## Proxies

It is sometimes more convenient to use a proxy, rather than a weak reference. Proxies can be used as though they were the original object, and do not need to be called before the object is accessible. As a consequence, they can be passed to a library that does not know it is receiving a reference instead of the real object.

```
# weakref_proxy.py

import weakref

class ExpensiveObject:

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

    def __del__(self):
        print('(Deleting {})'.format(self))

obj = ExpensiveObject('My Object')
r = weakref.ref(obj)
p = weakref.proxy(obj)

print('via obj:', obj.name)
print('via ref:', r().name)
print('via proxy:', p.name)
del obj
print('via proxy:', p.name)
```

If the proxy is accessed after the referent object is removed, a ReferenceError exception is raised.

```
$ python3 weakref_proxy.py

via obj: My Object
```

```

via ref: My Object
via proxy: My Object
(Deleting <__main__.ExpensiveObject object at 0x1007aa7b8>)
Traceback (most recent call last):
  File "weakref_proxy.py", line 30, in <module>
    print('via proxy:', p.name)
ReferenceError: weakly-referenced object no longer exists

```

## Caching Objects

The ref and proxy classes are considered “low level.” While they are useful for maintaining weak references to individual objects and allowing cycles to be garbage collected, the WeakKeyDictionary and WeakValueDictionary classes provide a more appropriate API for creating a cache of several objects.

The WeakValueDictionary class uses weak references to the values it holds, allowing them to be garbage collected when other code is not actually using them. Using explicit calls to the garbage collector illustrates the difference between memory handling with a regular dictionary and WeakValueDictionary:

```

# weakref_valuedict.py

import gc
from pprint import pprint
import weakref

gc.set_debug(gc.DEBUG_UNCOLLECTABLE)

class ExpensiveObject:

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

    def __repr__(self):
        return 'ExpensiveObject({})'.format(self.name)

    def __del__(self):
        print('    (Deleting {})'.format(self))

def demo(cache_factory):
    # hold objects so any weak references
    # are not removed immediately
    all_refs = {}
    # create the cache using the factory
    print('CACHE TYPE:', cache_factory)
    cache = cache_factory()
    for name in ['one', 'two', 'three']:
        o = ExpensiveObject(name)
        cache[name] = o
        all_refs[name] = o
        del o # decref

    print(' all_refs =', end=' ')
    pprint(all_refs)
    print('\n Before, cache contains:', list(cache.keys()))
    for name, value in cache.items():
        print('    {} = {}'.format(name, value))
        del value # decref

    # remove all references to the objects except the cache
    print('\n Cleanup:')
    del all_refs
    gc.collect()

    print('\n After, cache contains:', list(cache.keys()))
    for name, value in cache.items():
        print('    {} = {}'.format(name, value))
    print(' demo returning')
    return

```

```
demo(dict)
print()

demo(weakref.WeakValueDictionary)
```

Any loop variables that refer to the values being cached must be cleared explicitly so the reference count of the object is decremented. Otherwise, the garbage collector will not remove the objects and they will remain in the cache. Similarly, the `all_refs` variable is used to hold references to prevent them from being garbage collected prematurely.

```
$ python3 weakref_valuedict.py

CACHE TYPE: <class 'dict'>
all_refs = {'one': ExpensiveObject(one),
'three': ExpensiveObject(three),
'two': ExpensiveObject(two)}

Before, cache contains: ['one', 'three', 'two']
one = ExpensiveObject(one)
three = ExpensiveObject(three)
two = ExpensiveObject(two)

Cleanup:

After, cache contains: ['one', 'three', 'two']
one = ExpensiveObject(one)
three = ExpensiveObject(three)
two = ExpensiveObject(two)
demo returning
(Deleting ExpensiveObject(one))
(Deleting ExpensiveObject(three))
(Deleting ExpensiveObject(two))

CACHE TYPE: <class 'weakref.WeakValueDictionary'>
all_refs = {'one': ExpensiveObject(one),
'three': ExpensiveObject(three),
'two': ExpensiveObject(two)}

Before, cache contains: ['one', 'three', 'two']
one = ExpensiveObject(one)
three = ExpensiveObject(three)
two = ExpensiveObject(two)

Cleanup:
(Deleting ExpensiveObject(one))
(Deleting ExpensiveObject(three))
(Deleting ExpensiveObject(two))

After, cache contains: []
demo returning
```

The `WeakKeyDictionary` works similarly but uses weak references for the keys instead of the values in the dictionary.

## Warning

The library documentation for `weakref` contains this warning:

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).

## See also

- [Standard library documentation for weakref](#)
- [gc](#) – The `gc` module is the interface to the interpreter’s garbage collector.
- [PEP 205](#) – “Weak References” enhancement proposal.

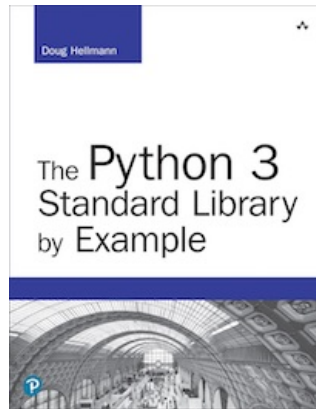
## Quick Links

[References](#)  
[Reference Callbacks](#)  
[Finalizing Objects](#)  
[Proxies](#)  
[Caching Objects](#)

*This page was last updated 2017-01-28.*

## Navigation

[→ struct — Binary Data Structures](#)  
[→ copy — Duplicate Objects](#)





[Get the book](#)

*The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.*

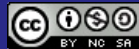
Looking for [examples for Python 2?](#)

## This Site

 [Module Index](#)  
 [Index](#)



© Copyright 2019, Doug Hellmann



## Other Writing

 [Blog](#)  
 [The Python Standard Library By Example](#)