Write More Robust Code with Weak References

Jim Baker

jim.baker@{python.org, rackspace.com}

Some possible questions

Questions you might have in coming to this talk:

- What exactly are weak references?

Some possible questions

Questions you might have in coming to this talk:

- What exactly are weak references?
- How do they differ from strong references?

Some possible questions

Questions you might have in coming to this talk:

- What exactly are weak references?
- How do they differ from strong references?
- When would I use them anyway?

About me

- Core developer of Jython

About me

- Core developer of Jython
- Co-author of *Definitive Guide to Jython* from Apress

About me

- Core developer of Jython
- Co-author of *Definitive Guide to Jython* from Apress
- Software developer at Rackspace

About me

- Core developer of Jython
- Co-author of *Definitive Guide to Jython* from Apress
- Software developer at Rackspace
- Lecturer in CS at Univ of Colorado at Boulder

Defining 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.*

(https://docs.python.org/3/library/weakref.html)

Weak references

- Initially proposed in PEP 205

Weak references

- Initially proposed in PEP 205
- Implemented in Python 2.1 (released April 2001)

Weak references

- Initially proposed in PEP 205
- Implemented in Python 2.1 (released April 2001)
- **Released 14 years ago!**

Example: `WeakSet`

First, let's import `WeakSet`. Many uses of weak references are with respect to the collections provided by the `weakref` module:

```
from weakref import WeakSet
```

Weak referenceable classes

Define a class `X` like so:

```
class X(object):
    pass
```

NB: `str` and certain other classes are not weak referenceable in CPython, but their subclasses can be

Construction

Construct a weak set and add an element to it. We then list the set:

```
s = WeakSet()
s.add(X())
list(s)
```

Conclusions

- s is (eventually) empty - with list(s), we get []

Conclusions

- `s` is (eventually) empty - with list(s), we get `[]`
- May require a round of garbage collection with `gc.collect()`

Some possible questions

Questions you might have in coming to this talk:

- What exactly are weak references?
- How do they differ from strong references?
- When would I use them anyway?

To prevent memory and resource leaks.

Resource leaks

Often you can write code like this, without explicitly calling `f.close()`:

```
f = open("foo.txt")
...
```

But not always...

Garbage collection is not magical

GC works by determining that some some set of objects is unreachable:

- Doesn't matter if it's reference counting

Garbage collection is not magical

GC works by determining that some some set of objects is unreachable:

- Doesn't matter if it's reference counting
- Or a variant of mark-and-sweep

Garbage collection is not magical

GC works by determining that some some set of objects is unreachable:

- Doesn't matter if it's reference counting
- Or a variant of mark-and-sweep
- Or the combination used by CPython, to account for reference cycles

Takeaway

- It cannot read your mind, developer though you may be!

Takeaway

- It cannot read your mind, developer though you may be!
- GC is not sufficient to manage the lifecycle of resources

Manual clearance

- Clean up resources - setting to `None`, calling `close()`, ...

Manual clearance

- Clean up resources - setting to `None`, calling `close()`, ...
- Use `try/finally`

try/finally

```
try:
    f = open("foo.txt")
    ...
finally:
    f.close()
```

Manual clearance

- Clean up resources - setting to `None`, calling `close()`, ...
- Use `try/finally`
- Apply deeper knowledge of your code

Manual clearance

- Clean up resources - setting to `None`, calling `close()`, ...
- Use `try/finally`
- Apply deeper knowledge of your code
- Or do cleanup by some other scheme

Finalizers with `__del__`

- May use finalizers because of explicit external resource management

Finalizers with `__del__`

- May use finalizers because of explicit external resource management
- Especially in conjunction with some explicit ref counting

socket.makefile

```
socket.makefile([mode[, bufsize]])
```

Return a file object associated with the socket. (File objects are described in File Objects.) The file object does not close the socket explicitly when its close() method is called, but only removes its reference to the socket object, so that the socket will be closed if it is not referenced from anywhere else.

errno.EMFILE?

Otherwise we may see an `IOError` raised with `errno.EMFILE` ("Too many open files")

socket.makefile

`socket.makefile([mode[, bufsize]])`

> *Return a file object associated with the socket. (File objects are described in File Objects.) The file object does not close the socket explicitly when its close() method is called, but only removes its reference to the socket object, so that the socket will be closed if it is not referenced from anywhere else.*

Implementation is done through a separate ref counting scheme

_fileobject

Prevent resource leaks (of underlying sockets) in the socket module:

```
class _fileobject(object):
  ...
  def __del__(self):
    try:
      self.close()
    except:
      # close() may fail if __init__ didn't complete
      pass
```

NB: changed in Python 3.x, above is 2.7 implementation

with statement for ARM

You are already using automatic resource management, right?

```
with open("foo.txt") as f:
    ...
```

So far, so good

- No weak references yet

So far, so good

- No weak references yet
- Keeping it simple!

So far, so good

- No weak references yet
- Keeping it simple!
- No need to be in this talk, right?

What if. . .

- An object is a child in a parent-child relationship?

What if. . .

- An object is a child in a parent-child relationship?
- And needs to track its parent?

What if. . .

- An object is a child in a parent-child relationship?
- And needs to track its parent?
- And the parent wants to track the child?

What if. . .

- An object is a child in a parent-child relationship?
- And needs to track its parent?
- And the parent wants to track the child?
- Example: `xml.sax.expatreader`

Make it even simpler

- Let's implement a doubly-linked list - next and previous references

Make it even simpler

- Let's implement a doubly-linked list - next and previous references
- But also add `__del__` to clean up resources

OrderedDict

- Dict that preserves the order of insertion, for iteration and indexed access

OrderedDict

- Dict that preserves the order of insertion, for iteration and indexed access
- Asymptotic performance (big-O running time) same as regular dicts

OrderedDict

- Dict that preserves the order of insertion, for iteration and indexed access
- Asymptotic performance (big-O running time) same as regular dicts
- Uses a doubly-linked list to preserve insertion order

Avoiding reference cycles

- Why is avoiding strong reference cycles important?

Avoiding reference cycles

- Why is avoiding strong reference cycles important?
- CPython's GC usually does reference counting

Avoiding reference cycles

- Why is avoiding strong reference cycles important?
- CPython's GC usually does reference counting
- But a cycle cannot go to **zero**

Under the hood

- CPython's weak reference scheme stores a list of containers to be cleared out, including proxies

Under the hood

- CPython's weak reference scheme stores a list of containers to be cleared out, including proxies
- Performed when the referred object is deallocated

Under the hood

- CPython's weak reference scheme stores a list of containers to be cleared out, including proxies
- Performed when the referred object is deallocated
- Which occurs when the refcount goes to zero

Under the hood

- CPython's weak reference scheme stores a list of containers to be cleared out, including proxies
- Performed when the referred object is deallocated
- Which occurs when the refcount goes to zero
- **No waiting on the garbage collector!**

Example: set

From `setobject.c` in CPython 3.5

```
static void
set_dealloc(PySetObject *so)
{
    setentry *entry;
    Py_ssize_t fill = so->fill;
    PyObject_GC_UnTrack(so);
    Py_TRASHCAN_SAFE_BEGIN(so)
    if (so->weakreflist != NULL)
        PyObject_ClearWeakRefs((PyObject *) so);
    ...
```

Also explains why many lightweight objects in CPython are not weak referenceable - avoid the cost of extra overhead of the `weakreflist`

Ref cycles using GC in CPython

- Strong reference cycles have to wait for mark-and-sweep GC

Ref cycles using GC in CPython

- Strong reference cycles have to wait for mark-and-sweep GC
- CPython's GC is stop-the-world

Ref cycles using GC in CPython

- Strong reference cycles have to wait for mark-and-sweep GC
- CPython's GC is stop-the-world
- Runs only per decision criteria in the `gc.set_threshold`, which is now generational

Ref cycles using GC in CPython

- Strong reference cycles have to wait for mark-and-sweep GC
- CPython's GC is stop-the-world
- Runs only per decision criteria in the `gc.set_threshold`, which is now generational
- Doesn't occur when you need it to close that file, or some other issue

Useful points to consider

- My experience with garbage collectors is that they work well, except when they don't

Useful points to consider

- My experience with garbage collectors is that they work well, except when they don't
- Especially around a small object pointing to an expensive resource

Useful points to consider

- My experience with garbage collectors is that they work well, except when they don't
- Especially around a small object pointing to an expensive resource
- Which you might see with resources that have limits

Bug!

http://bugs.python.org/issue9825

- For 2.7, removed `__del__` in r84725

Bug!

http://bugs.python.org/issue9825

- For 2.7, removed `__del__` in r84725
- For 3.2, replaced `__del__` with weakrefs in r84727

Bug!

http://bugs.python.org/issue9825

- For 2.7, removed `__del__` in r84725
- For 3.2, replaced `__del__` with weakrefs in r84727
- For 3.4, using `__del__` no longer means ref cycles are uncollectable garbage

Python 2.7 solution

> *Issue #9825: removed __del__ from the definition of collections.OrderedDict. This prevents user-created self-referencing ordered dictionaries from becoming permanently uncollectable GC garbage. The downside is that removing __del__ means that the internal doubly-linked list has to wait for GC collection rather than freeing memory immediately when the refcnt drops to zero.*

So this is an important fix - don't want uncollectable garbage!

Bug!

http://bugs.python.org/issue9825

- For 2.7, removed `__del__` in r84725
- For 3.2, replaced `__del__` with weakrefs in r84727

Bug!

http://bugs.python.org/issue9825

- For 2.7, removed `__del__` in r84725
- For 3.2, replaced `__del__` with weakrefs in r84727
- For 3.4, using `__del__` no longer means ref cycles are uncollectable garbage

Weak references to the rescue!

See implementation of `collections.OrderedDict`

Crux of the code

- Use `__slots__` to minimize overhead - no need for a dict per object here

```
__slots__ = 'prev', 'next', 'key', '__weakref__'
```

Crux of the code

- Use `__slots__` to minimize overhead - no need for a dict per object here
- `__weakref__` means that a slots-built class should be weak referenceable

```
__slots__ = 'prev', 'next', 'key', '__weakref__'
```

Crux of the code

- Use `__slots__` to minimize overhead - no need for a dict per object here
- `__weakref__` means that a slots-built class should be weak referenceable
- NB: no-op in implementations like Jython

```
__slots__ = 'prev', 'next', 'key', '__weakref__'
```

Crux of the code (2)


```
root.prev = proxy(link)
```

Lookup tables

- Want to provide more information about a given object

Lookup tables

- Want to provide more information about a given object
- Without extending/monkeypatching it

Lookup tables

- Want to provide more information about a given object
- Without extending/monkeypatching it
- (So no use of `__dict__` for extra properties)

Using a `dict`

- Could use the object as a key

Using a `dict`

- Could use the object as a key
- But need to manually clean up the `dict` when the object is no longer needed

Using a `dict`

- Could use the object as a key
- But need to manually clean up the `dict` when the object is no longer needed
- Maybe you know, maybe you don't. Especially useful for libraries

WeakKeyDictionary

- Insert the object as the key

WeakKeyDictionary

- Insert the object as the key
- Associate anything you want as a value - list of proprerties, another object, etc

WeakKeyDictionary

- Insert the object as the key
- Associate anything you want as a value - list of proprerties, another object, etc
- When the object used as key goes away, the value is also cleared out (if nothing else is holding onto it)

Example: Django signals

Django uses weak references in the implementation of its signal mechanism:

> *Django includes a "signal dispatcher" which helps allow decoupled applications get notified when actions occur elsewhere in the framework. In a nutshell, signals allow certain senders to notify a set of receivers that some action has taken place. They're especially useful when many pieces of code may be interested in the same events.*

WeakKeyDictionary

Avoid computing the senders-receivers coupling on the fly, the easy way:

```
self.sender_receivers_cache = weakref.WeakKeyDict
    if use_caching else {}
```

WeakValueDictionary

- Why?

WeakValueDictionary

- Why?
- Used by multiprocessing (track processes), logging (track handlers) , symtable. . .

WeakValueDictionary

- Why?
- Used by multiprocessing (track processes), logging (track handlers) , symtable. . .
- Useful for when you want to track the object by some id, and there should only be one, but once the object is no longer needed, you can let it go

Object lifecycle independence

- One side may depend on the other, but not vice versa

Object lifecycle independence

- One side may depend on the other, but not vice versa
- Use weak references for the independent side - process is terminated, can remove the lookup by process id

Object lifecycle independence

- One side may depend on the other, but not vice versa
- Use weak references for the independent side - process is terminated, can remove the lookup by process id
- -> `WeakValueDictionary`

Combining *both* weak keys and weak values?

Yes, it does make sense. Both sides are independent.

Example: Mapping Java classes to Python wrappers

Jython implements this variant of the Highlander pattern:

- Map the Java class to Python wrappers (strong ref from using Java code)

Example: Mapping Java classes to Python wrappers

Jython implements this variant of the Highlander pattern:

- Map the Java class to Python wrappers (strong ref from using Java code)
- Python classes to any using Java class (strong ref from using Python code)

Example: Mapping Java classes to Python wrappers

Jython implements this variant of the Highlander pattern:

- Map the Java class to Python wrappers (strong ref from using Java code)
- Python classes to any using Java class (strong ref from using Python code)
- AND *there can only be one* mapping (or at least should be)

Either might go away

- Why?

Either might go away

- Why?
- Java classes will be garbage collected if no `ClassLoader` (the parent of the class effectively) or objects of that class exist;

Either might go away

- Why?
- Java classes will be garbage collected if no `ClassLoader` (the parent of the class effectively) or objects of that class exist;
- But Python usage of this class will be GCed if no usage on the Python side - no subclasses in Python, etc

Implementations

- Pure Python Recipe available (http://code.activestate.com/recipes/528879-weak-key-and-value-dictionary/) but I haven't evaluated

Implementations

- Pure Python Recipe available (http://code.activestate.com/recipes/528879-weak-key-and-value-dictionary/) but I haven't evaluated
- Easy Jython version, because of JVM ecosystem

Jython version

```
from jythonlib import MapMaker, dict_builder

class WeakKeyValueDictionary(dict):
    def __new__(cls, *args, **kw):
        return WeakKeyValueDictionaryBuilder(*args, *

    # also add itervaluerefs, valuerefs,
    # iterkeyrefs, keyrefs
```

Hook into Google Guava Collections


```
WeakKeyValueDictionaryBuilder = dict_builder(
    MapMaker().weakKeys().weakValues().makeMap,
    WeakKeyValueDictionary)
```