gc — Garbage Collector

Purpose: Manages memory used by Python objects

gc exposes the underlying memory management mechanism of Python, the automatic garbage collector. The module includes functions for controlling how the collector operates and to examine the objects known to the system, either pending collection or stuck in reference cycles and unable to be freed.

Tracing References

With gc the incoming and outgoing references between objects can be used to find cycles in complex data structures. If a data structure is known to have a cycle, custom code can be used to examine its properties. If the cycle is in unknown code, the `get_referents()` and `get_referrers()` functions can be used to build generic debugging tools.

For example, `get_referents()` shows the objects *referred to* by the input arguments.

```
# gc_get_referents.py

import gc
import pprint


class Graph:

    def __init__(self, name):
        self.name = name
        self.next = None

    def set_next(self, next):
        print('Linking nodes {}.next = {}'.format(self, next))
        self.next = next

    def __repr__(self):
        return '{}({})'.format(
            self.__class__.__name__, self.name)


# Construct a graph cycle
one = Graph('one')
two = Graph('two')
three = Graph('three')
one.set_next(two)
two.set_next(three)
three.set_next(one)

print()
print('three refers to:')
for r in gc.get_referents(three):
    pprint.pprint(r)
```

In this case, the Graph instance three holds references to its instance dictionary (in the `__dict__` attribute) and its class.

```
$ python3 gc_get_referents.py

Linking nodes Graph(one).next = Graph(two)
Linking nodes Graph(two).next = Graph(three)
Linking nodes Graph(three).next = Graph(one)

three refers to:
{'name': 'three', 'next': Graph(one)}
<class '__main__.Graph'>
```

The next example uses a Queue to perform a breadth-first traversal of all of the object references looking for cycles. The items

inserted into the queue are tuples containing the reference chain so far and the next object to examine. It starts with `three`, and looks at everything it refers to. Skipping classes avoids looking at methods, modules, etc.

```
# gc_get_referents_cycles.py

import gc
import pprint
import queue


class Graph:

    def __init__(self, name):
        self.name = name
        self.next = None

    def set_next(self, next):
        print('Linking nodes {}.next = {}'.format(self, next))
        self.next = next

    def __repr__(self):
        return '{}({})'.format(
            self.__class__.__name__, self.name)


# Construct a graph cycle
one = Graph('one')
two = Graph('two')
three = Graph('three')
one.set_next(two)
two.set_next(three)
three.set_next(one)

print()

seen = set()
to_process = queue.Queue()

# Start with an empty object chain and Graph three.
to_process.put(([], three))

# Look for cycles, building the object chain for each object
# found in the queue so the full cycle can be printed at the
# end.
while not to_process.empty():
    chain, next = to_process.get()
    chain = chain[:]
    chain.append(next)
    print('Examining:', repr(next))
    seen.add(id(next))
    for r in gc.get_referents(next):
        if isinstance(r, str) or isinstance(r, type):
            # Ignore strings and classes
            pass
        elif id(r) in seen:
            print()
            print('Found a cycle to {}:'.format(r))
            for i, link in enumerate(chain):
                print('  {}: '.format(i), end=' ')
                pprint.pprint(link)
        else:
            to_process.put((chain, r))
```

The cycle in the nodes is easily found by watching for objects that have already been processed. To avoid holding references to those objects, their `id()` values are cached in a set. The dictionary objects found in the cycle are the `__dict__` values for the `Graph` instances, and hold their instance attributes.

```
$ python3 gc_get_referents_cycles.py

Linking nodes Graph(one).next = Graph(two)
Linking nodes Graph(two).next = Graph(three)
```

```
Linking nodes Graph(three).next = Graph(one)

Examining: Graph(three)
Examining: {'name': 'three', 'next': Graph(one)}
Examining: Graph(one)
Examining: {'name': 'one', 'next': Graph(two)}
Examining: Graph(two)
Examining: {'name': 'two', 'next': Graph(three)}

Found a cycle to Graph(three):
  0:  Graph(three)
  1:  {'name': 'three', 'next': Graph(one)}
  2:  Graph(one)
  3:  {'name': 'one', 'next': Graph(two)}
  4:  Graph(two)
  5:  {'name': 'two', 'next': Graph(three)}
```

Forcing Garbage Collection

Although the garbage collector runs automatically as the interpreter executes a program, it can be triggered to run at a specific time when there are a lot of objects to free or there is not much work happening and the collector will not hurt application performance. Trigger collection using `collect()`.

```
# gc_collect.py

import gc
import pprint


class Graph:

    def __init__(self, name):
        self.name = name
        self.next = None

    def set_next(self, next):
        print('Linking nodes {}.next = {}'.format(self, next))
        self.next = next

    def __repr__(self):
        return '{}({})'.format(
            self.__class__.__name__, self.name)


# Construct a graph cycle
one = Graph('one')
two = Graph('two')
three = Graph('three')
one.set_next(two)
two.set_next(three)
three.set_next(one)

# Remove references to the graph nodes in this module's namespace
one = two = three = None

# Show the effect of garbage collection
for i in range(2):
    print('\nCollecting {} ...'.format(i))
    n = gc.collect()
    print('Unreachable objects:', n)
    print('Remaining Garbage:', end=' ')
    pprint.pprint(gc.garbage)
```

In this example, the cycle is cleared as soon as collection runs the first time, since nothing refers to the `Graph` nodes except themselves. `collect()` returns the number of “unreachable” objects it found. In this case, the value is 6 because there are three objects with their instance attribute dictionaries.

```
$ python3 gc_collect.py
```

```
Linking nodes Graph(one).next = Graph(two)
Linking nodes Graph(two).next = Graph(three)
Linking nodes Graph(three).next = Graph(one)

Collecting 0 ...
Unreachable objects: 6
Remaining Garbage: []

Collecting 1 ...
Unreachable objects: 0
Remaining Garbage: []
```

Finding References to Objects that Cannot be Collected

Looking for the object holding a reference to another object is a little trickier than seeing what an object references. Because the code asking about the reference needs to hold a reference itself, some of the referrers need to be ignored. This example creates a graph cycle, then works through the `Graph` instances and removes the reference in the “parent” node.

```
# gc_get_referrers.py

import gc
import pprint


class Graph:

    def __init__(self, name):
        self.name = name
        self.next = None

    def set_next(self, next):
        print('Linking nodes {}.next = {}'.format(self, next))
        self.next = next

    def __repr__(self):
        return '{}({})'.format(
            self.__class__.__name__, self.name)

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


# Construct a graph cycle
one = Graph('one')
two = Graph('two')
three = Graph('three')
one.set_next(two)
two.set_next(three)
three.set_next(one)

# Collecting now keeps the objects as uncollectable,
# but not garbage.
print()
print('Collecting...')
n = gc.collect()
print('Unreachable objects:', n)
print('Remaining Garbage:', end=' ')
pprint.pprint(gc.garbage)

# Ignore references from local variables in this module, global
# variables, and from the garbage collector's bookkeeping.
REFERRERS_TO_IGNORE = [locals(), globals(), gc.garbage]


def find_referring_graphs(obj):
    print('Looking for references to {!r}'.format(obj))
    referrers = (r for r in gc.get_referrers(obj)
                 if r not in REFERRERS_TO_IGNORE)
    for ref in referrers:
        if isinstance(ref, Graph):
```

```
            # A graph node
            yield ref
        elif isinstance(ref, dict):
            # An instance or other namespace dictionary
            for parent in find_referring_graphs(ref):
                yield parent


# Look for objects that refer to the objects in the graph.
print()
print('Clearing referrers:')
for obj in [one, two, three]:
    for ref in find_referring_graphs(obj):
        print('Found referrer:', ref)
        ref.set_next(None)
        del ref  # remove reference so the node can be deleted
    del obj  # remove reference so the node can be deleted

# Clear references held by gc.garbage
print()
print('Clearing gc.garbage:')
del gc.garbage[:]

# Everything should have been freed this time
print()
print('Collecting...')
n = gc.collect()
print('Unreachable objects:', n)
print('Remaining Garbage:', end=' ')
pprint.pprint(gc.garbage)
```

This sort of logic is overkill if the cycles are understood, but for an unexplained cycle in data using `get_referrers()` can expose the unexpected relationship.

```
$ python3 gc_get_referrers.py

Linking nodes Graph(one).next = Graph(two)
Linking nodes Graph(two).next = Graph(three)
Linking nodes Graph(three).next = Graph(one)

Collecting...
Unreachable objects: 0
Remaining Garbage: []

Clearing referrers:
Looking for references to Graph(one)
Looking for references to {'name': 'three', 'next': Graph(one)}
Found referrer: Graph(three)
Linking nodes Graph(three).next = None
Looking for references to Graph(two)
Looking for references to {'name': 'one', 'next': Graph(two)}
Found referrer: Graph(one)
Linking nodes Graph(one).next = None
Looking for references to Graph(three)
Looking for references to {'name': 'two', 'next': Graph(three)}
Found referrer: Graph(two)
Linking nodes Graph(two).next = None

Clearing gc.garbage:

Collecting...
Unreachable objects: 0
Remaining Garbage: []
Graph(one).__del__()
Graph(two).__del__()
Graph(three).__del__()
```

Collection Thresholds and Generations

The garbage collector maintains three lists of objects it sees as it runs, one for each “generation” tracked by the collector. As

objects are examined in each generation, they are either collected or they age into subsequent generations until they finally reach the stage where they are kept permanently.

The collector routines can be tuned to occur at different frequencies based on the difference between the number of object allocations and deallocations between runs. When the number of allocations minus the number of deallocations is greater than the threshold for the generation, the garbage collector is run. The current thresholds can be examined with `get_threshold()`.

```
# gc_get_threshold.py

import gc

print(gc.get_threshold())
```

The return value is a tuple with the threshold for each generation.

```
$ python3 gc_get_threshold.py

(700, 10, 10)
```

The thresholds can be changed with `set_threshold()`. This example program uses a command line argument to set the threshold for generation `0` then allocates a series of objects.

```
# gc_threshold.py

import gc
import pprint
import sys

try:
    threshold = int(sys.argv[1])
except (IndexError, ValueError, TypeError):
    print('Missing or invalid threshold, using default')
    threshold = 5


class MyObj:

    def __init__(self, name):
        self.name = name
        print('Created', self.name)


gc.set_debug(gc.DEBUG_STATS)

gc.set_threshold(threshold, 1, 1)
print('Thresholds:', gc.get_threshold())

print('Clear the collector by forcing a run')
gc.collect()
print()

print('Creating objects')
objs = []
for i in range(10):
    objs.append(MyObj(i))
print('Exiting')

# Turn off debugging
gc.set_debug(0)
```

Different threshold values introduce the garbage collection sweeps at different times, shown here because debugging is enabled.

```
$ python3 -u gc_threshold.py 5

Thresholds: (5, 1, 1)
Clear the collector by forcing a run
gc: collecting generation 2...
gc: objects in each generation: 575 1279 4527
gc: objects in permanent generation: 0
```

```
gc: objects in permanent generation: 0
gc: done, 0.0009s elapsed

Creating objects
gc: collecting generation 0...
gc: objects in each generation: 3 0 6137
gc: objects in permanent generation: 0
gc: done, 0.0000s elapsed
Created 0
Created 1
Created 2
gc: collecting generation 0...
gc: objects in each generation: 3 2 6137
gc: objects in permanent generation: 0
gc: done, 0.0000s elapsed
Created 3
Created 4
Created 5
gc: collecting generation 1...
gc: objects in each generation: 4 4 6137
gc: objects in permanent generation: 0
gc: done, 0.0000s elapsed
Created 6
Created 7
Created 8
gc: collecting generation 0...
gc: objects in each generation: 4 0 6144
gc: objects in permanent generation: 0
gc: done, 0.0000s elapsed
Created 9
Exiting
```

A smaller threshold causes the sweeps to run more frequently.

```
$ python3 -u gc_threshold.py 2

Thresholds: (2, 1, 1)
Clear the collector by forcing a run
gc: collecting generation 2...
gc: objects in each generation: 575 1279 4527
gc: objects in permanent generation: 0
gc: done, 0.0026s elapsed
gc: collecting generation 0...
gc: objects in each generation: 1 0 6137
gc: objects in permanent generation: 0
gc: done, 0.0000s elapsed

Creating objects
gc: collecting generation 0...
gc: objects in each generation: 3 0 6137
gc: objects in permanent generation: 0
gc: done, 0.0000s elapsed
Created 0
gc: collecting generation 1...
gc: objects in each generation: 2 2 6137
gc: objects in permanent generation: 0
gc: done, 0.0000s elapsed
Created 1
Created 2
gc: collecting generation 0...
gc: objects in each generation: 2 0 6140
gc: objects in permanent generation: 0
gc: done, 0.0000s elapsed
Created 3
gc: collecting generation 0...
gc: objects in each generation: 3 1 6140
gc: objects in permanent generation: 0
gc: done, 0.0000s elapsed
Created 4
Created 5
gc: collecting generation 1...
gc: objects in each generation: 2 3 6140
```

```
gc: objects in permanent generation: 0
gc: done, 0.0000s elapsed
Created 6
gc: collecting generation 0...
gc: objects in each generation: 3 0 6144
gc: objects in permanent generation: 0
gc: done, 0.0000s elapsed
Created 7
Created 8
gc: collecting generation 0...
gc: objects in each generation: 2 2 6144
gc: objects in permanent generation: 0
gc: done, 0.0000s elapsed
Created 9
Exiting
```

Debugging

Debugging memory leaks can be challenging. `gc` includes several options to expose the inner workings to make the job easier. The options are bit-flags meant to be combined and passed to `set_debug()` to configure the garbage collector while the program is running. Debugging information is printed to `sys.stderr`.

The `DEBUG_STATS` flag turns on statistics reporting, causing the garbage collector to report when it is running, the number of objects tracked for each generation, and the amount of time it took to perform the sweep.

```
# gc_debug_stats.py

import gc

gc.set_debug(gc.DEBUG_STATS)

gc.collect()
print('Exiting')
```

This example output shows two separate runs of the collector because it runs once when it is invoked explicitly, and a second time when the interpreter exits.

```
$ python3 gc_debug_stats.py

gc: collecting generation 2...
gc: objects in each generation: 826 471 4529
gc: objects in permanent generation: 0
gc: done, 24 unreachable, 0 uncollectable, 0.0007s elapsed
Exiting
gc: collecting generation 2...
gc: objects in each generation: 1 0 5552
gc: objects in permanent generation: 0
gc: done, 0.0005s elapsed
gc: collecting generation 2...
gc: objects in each generation: 107 0 5382
gc: objects in permanent generation: 0
gc: done, 1406 unreachable, 0 uncollectable, 0.0008s elapsed
gc: collecting generation 2...
gc: objects in each generation: 0 0 3307
gc: objects in permanent generation: 0
gc: done, 151 unreachable, 0 uncollectable, 0.0002s elapsed
```

Enabling `DEBUG_COLLECTABLE` and `DEBUG_UNCOLLECTABLE` causes the collector to report on whether each object it examines can or cannot be collected. If seeing the objects that cannot be collected is not enough information to understand where data is being retained, enable `DEBUG_SAVEALL` to cause `gc` to preserve all objects it finds without any references in the `garbage` list.

```
# gc_debug_saveall.py

import gc

flags = (gc.DEBUG_COLLECTABLE |
         gc.DEBUG_UNCOLLECTABLE |
         gc.DEBUG_SAVEALL
         )
```

```
        )

gc.set_debug(flags)


class Graph:

    def __init__(self, name):
        self.name = name
        self.next = None

    def set_next(self, next):
        self.next = next

    def __repr__(self):
        return '{}({})'.format(
            self.__class__.__name__, self.name)


class CleanupGraph(Graph):

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


# Construct a graph cycle
one = Graph('one')
two = Graph('two')
one.set_next(two)
two.set_next(one)

# Construct another node that stands on its own
three = CleanupGraph('three')

# Construct a graph cycle with a finalizer
four = CleanupGraph('four')
five = CleanupGraph('five')
four.set_next(five)
five.set_next(four)

# Remove references to the graph nodes in this module's namespace
one = two = three = four = five = None

# Force a sweep
print('Collecting')
gc.collect()
print('Done')

# Report on what was left
for o in gc.garbage:
    if isinstance(o, Graph):
        print('Retained: {} 0x{:x}'.format(o, id(o)))

# Reset the debug flags before exiting to avoid dumping a lot
# of extra information and making the example output more
# confusing.
gc.set_debug(0)
```

This allows the objects to be examined after garbage collection, which is helpful if, for example, the constructor cannot be changed to print the object id when each object is created.

```
$ python3 -u gc_debug_saveall.py

CleanupGraph(three).__del__()
Collecting
gc: collectable <tuple 0x1098186d0>
gc: collectable <cell 0x10983c3a8>
gc: collectable <tuple 0x109860da0>
gc: collectable <function 0x109864048>
gc: collectable <function 0x1098640d0>
gc: collectable <cell 0x10983c3d8>
gc: collectable <function 0x1098641e0>
```

```
gc: collectable <function 0x1098641c0>
gc: collectable <cell 0x10983c408>
gc: collectable <cell 0x10983c438>
gc: collectable <set 0x109854588>
gc: collectable <tuple 0x109838dc8>
gc: collectable <function 0x109864268>
gc: collectable <function 0x109864158>
gc: collectable <function 0x1098642f0>
gc: collectable <tuple 0x109860dd8>
gc: collectable <dict 0x10985c090>
gc: collectable <type 0x7fe2aa037a18>
gc: collectable <staticmethod 0x109860e10>
gc: collectable <member_descriptor 0x10985c8b8>
gc: collectable <member_descriptor 0x10985c900>
gc: collectable <member_descriptor 0x10985c948>
gc: collectable <getset_descriptor 0x10985c990>
gc: collectable <getset_descriptor 0x10985c9d8>
gc: collectable <tuple 0x109852b48>
gc: collectable <Graph 0x109771828>
gc: collectable <Graph 0x10979f320>
gc: collectable <dict 0x109745168>
gc: collectable <dict 0x109745318>
gc: collectable <CleanupGraph 0x1097af4e0>
gc: collectable <CleanupGraph 0x10983eb70>
gc: collectable <dict 0x1097ab558>
gc: collectable <dict 0x109869af8>
CleanupGraph(four).__del__()
CleanupGraph(five).__del__()
Done
Retained: Graph(one) 0x109771828
Retained: Graph(two) 0x10979f320
Retained: CleanupGraph(four) 0x1097af4e0
Retained: CleanupGraph(five) 0x10983eb70
```

For simplicity, DEBUG_LEAK is defined as a combination of all of the other options.

```
# gc_debug_leak.py

import gc

flags = gc.DEBUG_LEAK

gc.set_debug(flags)


class Graph:

    def __init__(self, name):
        self.name = name
        self.next = None

    def set_next(self, next):
        self.next = next

    def __repr__(self):
        return '{}({})'.format(
            self.__class__.__name__, self.name)


class CleanupGraph(Graph):

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


# Construct a graph cycle
one = Graph('one')
two = Graph('two')
one.set_next(two)
two.set_next(one)

# Construct another node that stands on its own
```

```
# Construct another node that stands on its own
three = CleanupGraph('three')

# Construct a graph cycle with a finalizer
four = CleanupGraph('four')
five = CleanupGraph('five')
four.set_next(five)
five.set_next(four)

# Remove references to the graph nodes in this module's namespace
one = two = three = four = five = None

# Force a sweep
print('Collecting')
gc.collect()
print('Done')

# Report on what was left
for o in gc.garbage:
    if isinstance(o, Graph):
        print('Retained: {} 0x{:x}'.format(o, id(o)))

# Reset the debug flags before exiting to avoid dumping a lot
# of extra information and making the example output more
# confusing.
gc.set_debug(0)
```

Keep in mind that because DEBUG_SAVEALL is enabled by DEBUG_LEAK, even the unreferenced objects that would normally have been collected and deleted are retained.

```
$ python3 -u gc_debug_leak.py

CleanupGraph(three).__del__()
Collecting
gc: collectable <tuple 0x10beb76d0>
gc: collectable <cell 0x10bedb3a8>
gc: collectable <tuple 0x10befada0>
gc: collectable <function 0x10befe048>
gc: collectable <function 0x10befe0d0>
gc: collectable <cell 0x10bedb3d8>
gc: collectable <function 0x10befe1e0>
gc: collectable <cell 0x10bedb408>
gc: collectable <cell 0x10bedb438>
gc: collectable <set 0x10bef3588>
gc: collectable <tuple 0x10bed7dc8>
gc: collectable <function 0x10befe268>
gc: collectable <function 0x10befe158>
gc: collectable <function 0x10befe2f0>
gc: collectable <tuple 0x10befadd8>
gc: collectable <dict 0x10bef6090>
gc: collectable <type 0x7ff77183ae18>
gc: collectable <staticmethod 0x10befae10>
gc: collectable <member_descriptor 0x10bef68b8>
gc: collectable <member_descriptor 0x10bef6900>
gc: collectable <member_descriptor 0x10bef6948>
gc: collectable <getset_descriptor 0x10bef6990>
gc: collectable <getset_descriptor 0x10bef69d8>
gc: collectable <tuple 0x10bef1b48>
gc: collectable <Graph 0x10be10828>
gc: collectable <Graph 0x10be3e320>
gc: collectable <dict 0x10bde4168>
gc: collectable <dict 0x10bde4318>
gc: collectable <CleanupGraph 0x10be4e4e0>
gc: collectable <CleanupGraph 0x10beddb70>
gc: collectable <dict 0x10be4a288>
gc: collectable <dict 0x10bf03af8>
CleanupGraph(four).__del__()
CleanupGraph(five).__del__()
Done
Retained: Graph(one) 0x10be10828
Retained: Graph(two) 0x10be3e320
Retained: CleanupGraph(four) 0x10be4e4e0
```

```
Retained: CleanupGraph(four) 0x10be4e4e0
Retained: CleanupGraph(five) 0x10beddb70
```

See also

- Standard library documentation for gc
- Python 2 to 3 porting notes for gc
- weakref – The weakref module provides a way to create references to objects without increasing their reference count, so they can still be garbage collected.
- Supporting Cyclic Garbage Collection – Background material from Python's C API documentation.
- How does Python manage memory? – An article on Python memory management by Fredrik Lundh.

resource — System Resource Management

sysconfig — Interpreter Compile-time Configuration

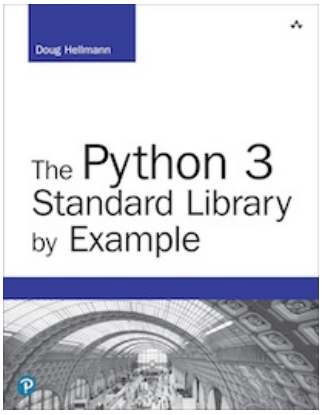
Quick Links

Tracing References
Forcing Garbage Collection
Finding References to Objects that Cannot be Collected
Collection Thresholds and Generations
Debugging

This page was last updated 2018-12-09.

Navigation

resource — System Resource Management
sysconfig — Interpreter Compile-time Configuration



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