28.12. `gc` — Garbage Collector interface

This module provides an interface to the optional garbage collector. It provides the ability to disable the collector, tune the collection frequency, and set debugging options. It also provides access to unreachable objects that the collector found but cannot free. Since the collector supplements the reference counting already used in Python, you can disable the collector if you are sure your program does not create reference cycles. Automatic collection can be disabled by calling `gc.disable()`. To debug a leaking program call `gc.set_debug(gc.DEBUG_LEAK)`. Notice that this includes `gc.DEBUG_SAVEALL`, causing garbage-collected objects to be saved in gc.garbage for inspection.

The `gc` module provides the following functions:

`gc.`**`enable`**`()`

Enable automatic garbage collection.

`gc.`**`disable`**`()`

Disable automatic garbage collection.

`gc.`**`isenabled`**`()`

Returns true if automatic collection is enabled.

`gc.`**`collect`**([*generation*])

With no arguments, run a full collection. The optional argument *generation* may be an integer specifying which generation to collect (from 0 to 2). A `ValueError` is raised if the generation number is invalid. The number of unreachable objects found is returned.

Changed in version 2.5: The optional *generation* argument was added.

Changed in version 2.6: The free lists maintained for a number of built-in types are cleared whenever a full collection or collection of the highest generation (2) is run. Not all items in some free lists may be freed due to the particular implementation, in particular `int` and `float`.

`gc.`**`set_debug`**(*flags*)

Set the garbage collection debugging flags. Debugging information will be written to `sys.stderr`. See below for a list of debugging flags which can be combined using bit operations to control debugging.

`gc.`**`get_debug`**`()`

Return the debugging flags currently set.

`gc.`**`get_objects`**`()`

Returns a list of all objects tracked by the collector, excluding the list returned.

New in version 2.2.

`gc.`**`set_threshold`**(*threshold0*[, *threshold1*[, *threshold2*]])

Set the garbage collection thresholds (the collection frequency). Setting *threshold0* to zero disables collection.

The GC classifies objects into three generations depending on how many collection sweeps they have survived. New objects are placed in the youngest generation (generation `0`). If an object survives a collection it is moved into the next older generation. Since generation `2` is the oldest generation, objects in that generation remain there after a collection. In order to decide when to run, the collector keeps track of the number object allocations and deallocations since the last collection. When the number of allocations minus the number of deallocations exceeds *threshold0*, collection starts. Initially only generation `0` is examined. If generation `0` has been examined more than *threshold1* times since generation `1` has been examined, then generation `1` is examined as well. Similarly, *threshold2* controls the number of collections of generation `1` before collecting generation `2`.

`gc.`**`get_count`**`()`

Return the current collection counts as a tuple of `(count0, count1, count2)`.

New in version 2.5.

`gc.`**`get_threshold`**`()`

Return the current collection thresholds as a tuple of `(threshold0, threshold1, threshold2)`.

`gc.`**`get_referrers`**`(`*`*objs`*`)`

Return the list of objects that directly refer to any of objs. This function will only locate those containers which support garbage collection; extension types which do refer to other objects but do not support garbage collection will not be found.

Note that objects which have already been dereferenced, but which live in cycles and have not yet been collected by the garbage collector can be listed among the resulting referrers. To get only currently live objects, call `collect()` before calling `get_referrers()`.

Care must be taken when using objects returned by `get_referrers()` because some of them could still be under construction and hence in a temporarily invalid state. Avoid using `get_referrers()` for any purpose other than debugging.

New in version 2.2.

`gc.`**`get_referents`**`(`*`*objs`*`)`

Return a list of objects directly referred to by any of the arguments. The referents returned are those objects visited by the arguments' C-level `tp_traverse` methods (if any), and may not be all objects actually directly reachable. `tp_traverse` methods are supported only by objects that support garbage collection, and are only required to visit objects that may be involved in a cycle. So, for example, if an integer is directly reachable from an argument, that integer object may or may not appear in the result list.

New in version 2.3.

`gc.`**`is_tracked`**`(`*`obj`*`)`

Returns `True` if the object is currently tracked by the garbage collector, `False` otherwise. As a general rule, instances of atomic types aren't tracked and instances of non-atomic types (containers, user-defined objects...) are. However, some type-specific optimizations can be present in order to suppress the garbage collector footprint of simple instances (e.g. dicts containing only atomic keys and values):

```
>>> gc.is_tracked(0)
False
>>> gc.is_tracked("a")
False
>>> gc.is_tracked([])
True
>>> gc.is_tracked({})
False
>>> gc.is_tracked({"a": 1})
False
>>> gc.is_tracked({"a": []})
True
```

New in version 2.7.

The following variable is provided for read-only access (you can mutate its value but should not rebind it):

`gc.`**`garbage`**

A list of objects which the collector found to be unreachable but could not be freed (uncollectable objects). By default, this list contains only objects with `__del__()` methods. [1] Objects that have `__del__()` methods and are part of a reference cycle cause the entire reference cycle to be uncollectable, including objects not necessarily in the cycle but reachable only from it. Python doesn't collect such cycles automatically because, in general, it isn't possible for Python to guess a safe order in which to run the `__del__()` methods. If you know a safe order, you can force the issue by examining the *garbage* list, and explicitly breaking cycles due to your objects within the list. Note that these objects are kept alive even so by virtue of being in the *garbage* list, so they should be removed from *garbage* too. For example, after breaking cycles, do `del gc.garbage[:]` to empty the list. It's generally better to avoid the issue by not creating cycles containing objects with `__del__()` methods, and *garbage* can be examined in that case to verify that no such cycles are being created.

If `DEBUG_SAVEALL` is set, then all unreachable objects will be added to this list rather than freed.

The following constants are provided for use with `set_debug()`:

`gc.`**`DEBUG_STATS`**

Print statistics during collection. This information can be useful when tuning the collection frequency.

`gc.`**`DEBUG_COLLECTABLE`**

Print information on collectable objects found.

`gc.`**`DEBUG_UNCOLLECTABLE`**

Print information of uncollectable objects found (objects which are not reachable but cannot be freed by the collector). These objects will be added to the `garbage` list.

`gc.`**`DEBUG_INSTANCES`**

When `DEBUG_COLLECTABLE` or `DEBUG_UNCOLLECTABLE` is set, print information about instance objects found.

gc.**DEBUG_OBJECTS**

When `DEBUG_COLLECTABLE` or `DEBUG_UNCOLLECTABLE` is set, print information about objects other than instance objects found.

gc.**DEBUG_SAVEALL**

When set, all unreachable objects found will be appended to *garbage* rather than being freed. This can be useful for debugging a leaking program.

gc.**DEBUG_LEAK**

The debugging flags necessary for the collector to print information about a leaking program (equal to `DEBUG_COLLECTABLE | DEBUG_UNCOLLECTABLE | DEBUG_INSTANCES | DEBUG_OBJECTS | DEBUG_SAVEALL`).

Footnotes

[1] Prior to Python 2.2, the list contained all instance objects in unreachable cycles, not only those with `__del__()` methods.