13. Controlling Execution - Python in a Nutshell, 3rd Edition

Garbage Collection

Python's garbage collection normally proceeds transparently and automatically, but you can choose to exert some direct control. The general principle is that Python collects each object `x` at some time after `x` becomes unreachable—that is, when no chain of references can reach `x` by starting from a local variable of a function instance that is executing, nor from a global variable of a loaded module. Normally, an object `x` becomes unreachable when there are no references at all to `x`. In addition, a group of objects can be unreachable when they reference each other but no global or local variables reference any of them, even indirectly (such a situation is known as a *mutual reference loop*).

Classic Python keeps with each object `x` a count, known as a *reference count*, of how many references to `x` are outstanding. When `x`'s reference count drops to `0`, CPython immediately collects `x`. The function `getrefcount` of the module `sys` accepts any object and returns its reference count (at least `1`, since `getrefcount` itself has a reference to the object it's examining). Other versions of Python, such as Jython or IronPython, rely on other garbage-collection mechanisms supplied by the platform they run on (e.g., the JVM or the MSCLR). The modules `gc` and `weakref` therefore apply only to CPython.

When Python garbage-collects `x` and there are no references at all to `x`, Python then finalizes `x` (i.e., calls `x.__del__()`) and makes the memory that `x` occupied available for other uses. If `x` held any references to other objects, Python removes the references, which in turn may make other objects collectable by leaving them unreachable.

The gc Module

The `gc` module exposes the functionality of Python's garbage collector. `gc` deals only with unreachable objects that are part of mutual reference loops. In such a loop, each object in the loop refers to others, keeping the reference counts of all objects positive. However, no outside references to any one of the set of mutually referencing objects exist any longer. Therefore, the whole group, also known as *cyclic garbage*, is unreachable, and therefore garbage-collectable. Looking for such cyclic garbage loops takes time, which is why the module `gc` exists: to help you control whether and when your program spends that time. The functionality of "cyclic garbage collection," by default, is enabled with some reasonable default parameters: however, by importing the `gc` module and calling its functions, you may choose to disable the functionality, change its parameters, and/or find out exactly what's going on in this respect.

`gc` exposes functions you can use to help you keep cyclic garbage-collection times under control. These functions can sometimes let you track down a memory leak—objects that are not getting collected even though there *should* be no more references to them—by helping you discover what other objects are in fact holding on to references to them:

collect	`collect()` Forces a full cyclic garbage collection run to happen immediately.
disable	`disable()` Suspends automatic, periodic cyclic garbage collection.

enable	`enable()` Reenables periodic cyclic garbage collection previously suspended with `disable`.
garbage	A read-only attribute that lists the unreachable but uncollectable objects. This happens when any object in a cyclic garbage loop has a `__del__` special method, as there may be no demonstrably safe order for Python to finalize such objects.
get_debug	`get_debug()` Returns an `int` bit string, the garbage-collection debug flags set with `set_debug`.
get_objects	`get_objects()` Returns a list of all objects currently tracked by the cyclic garbage collector.
get_referrers	`get_referrers(*objs)` Returns a list of all container objects, currently tracked by the cyclic garbage collector, that refer to any one or more of the arguments.
get_threshold	`get_threshold()` Returns a three-item tuple `(thresh0, thresh1, thresh2)`, the garbage-collection thresholds set with `set_threshold`.
isenabled	`isenabled()` Returns `True` when cyclic garbage collection is currently enabled. Returns `False` when collection is currently disabled.

set_debug `set_debug(flags)`

Sets debugging flags for garbage collection. `flags` is an `int`, interpreted as a bit string, built by ORing (with the bitwise-OR operator `|`) zero or more constants supplied by the module `gc`:

`DEBUG_COLLECTABLE`

Prints information on collectable objects found during collection

`DEBUG_INSTANCES`

If `DEBUG_COLLECTABLE` or `DEBUG_UNCOLLECTABLE` is also set, prints information on objects found during collection that are instances of old-style classes (v2 only)

`DEBUG_LEAK`

The set of debugging flags that make the garbage collector print all information that can help you diagnose memory leaks; same as the bitwise-OR of all other constants (except `DEBUG_STATS`, which serves a different purpose)

`DEBUG_OBJECTS`

If `DEBUG_COLLECTABLE` or `DEBUG_UNCOLLECTABLE` is also set, prints information on objects found during collection that are not instances of old-style classes (v2 only)

`DEBUG_SAVEALL`

Saves all collectable objects to the list `gc.garbage` (where uncollectable ones are also always saved) to help you diagnose leaks

`DEBUG_STATS`

Prints statistics during collection to help you tune the thresholds

`DEBUG_UNCOLLECTABLE`

Prints information on uncollectable objects found during collection

set_threshold `set_threshold(thresh0[,thresh1[,thresh2]])`

Sets thresholds that control how often cyclic garbage-collection cycles run. A `thresh0` of `0` disables garbage collection. Garbage collection is an advanced, specialized topic, and the details of the generational garbage-collection approach used in Python (and consequently the detailed meanings of these thresholds) are beyond the scope of this book; see the online docs for some details.

When you know there are no cyclic garbage loops in your program, or when you can't afford the delay of cyclic garbage collection at some crucial time, suspend automatic garbage collection by calling `gc.disable()`. You can enable collection again later by calling `gc.enable()`. You can test whether automatic collection is currently enabled by calling `gc.isenabled()`, which returns `True` or `False`. To control *when* time is spent collecting, you can call `gc.collect()` to force a full cyclic collection run to happen immediately. To wrap some time-critical code:

```
import gc
gc_was_enabled = gc.isenabled()
if gc_was_enabled:
    gc.collect()
    gc.disable()
# insert some time-critical code
here
if gc_was_enabled:
    gc.enable()
```

Other functionality in the module `gc` is more advanced and rarely used, and can be grouped into two areas. The functions `get_threshold` and `set_threshold` and debug flag `DEBUG_STATS` help you fine-tune garbage collection to optimize your program's performance. The rest of `gc`'s functionality can help you diagnose memory leaks in your program. While `gc` itself can automatically fix many leaks (as long as you avoid defining `__del__` in your classes, since the existence of `__del__` can block cyclic garbage collection), your program runs faster if it avoids creating cyclic garbage in the first place.

The weakref Module

Careful design can often avoid reference loops. However, at times you need objects to know about each other, and avoiding mutual references would distort and complicate your design. For example, a container has references to its items, yet it can often be useful for an object to know about a container holding it. The result is a reference loop: due to the mutual references, the container and items keep each other alive, even when all other objects forget about them. Weak references solve this problem by allowing objects to reference others without keeping them alive.

A *weak reference* is a special object `w` that refers to some other object `x` without incrementing `x`'s reference count. When `x`'s reference count goes down to `0`, Python finalizes and collects `x`, then informs `w` of `x`'s demise. Weak reference `w` can now either disappear or get marked as invalid in a controlled way. At any time, a given `w` refers to either the same object `x` as when `w` was created, or to nothing at all; a weak reference is never retargeted. Not all types of objects support being the target `x` of a weak reference `w`, but classes, instances, and functions do.

The `weakref` module exposes functions and types to create and manage weak references:

getweakrefcount	`getweakrefcount(x)` Returns `len(getweakrefs(x))`.
getweakrefs	`getweakrefs(x)` Returns a list of all weak references and proxies whose target is `x`.
proxy	`proxy(x[,f])` Returns a weak proxy `p` of type `ProxyType` (`CallableProxyType` when `x` is callable) with object `x` as the target. In most contexts, using `p` is just like using `x`, except that when you use `p` after `x` has been deleted, Python raises `ReferenceError`. `p` is not hashable (thus, you cannot use `p` as a dictionary key), even when `x` is. When `f` is present, it must be callable with one argument and is the finalization callback for `p` (i.e., right before finalizing `x`, Python calls `f(p)`). `f` executes right *after* `x` is no longer reachable from `p`.

ref	`ref(x[,f])` Returns a weak reference `w` of type `ReferenceType` with object `x` as the target. `w` is callable without arguments: calling `w()` returns `x` when `x` is still alive; otherwise, `w()` returns `None`. `w` is hashable when `x` is hashable. You can compare weak references for equality (`==`, `!=`), but not for order (`<`, `>`, `<=`, `>=`). Two weak references `x` and `y` are equal when their targets are alive and equal, or when `x is y`. When `f` is present, it must be callable with one argument and is the finalization callback for `w` (i.e., right before finalizing `x`, Python calls `f(w)`). `f` executes right *after* `x` is no longer reachable from `w`.
WeakKeyDictionary	`class WeakKeyDictionary(adict={})` A `WeakKeyDictionary d` is a mapping weakly referencing its keys. When the reference count of a key `k` in `d` goes to `0`, item `d[k]` disappears. `adict` is used to initialize the mapping.
WeakValueDictionary	`class WeakValueDictionary(adict={})` A `WeakValueDictionary d` is a mapping weakly referencing its values. When the reference count of a value `v` in `d` goes to `0`, all items of `d` such that `d[k] is v` disappear. `adict` is used to initialize the mapping.

`WeakKeyDictionary` lets you noninvasively associate additional data with some hashable objects, with no change to the objects. `WeakValueDictionary` lets you non-invasively record transient associations between objects, and build caches. In each case, use a weak mapping, rather than a `dict`, to ensure that an object that is otherwise garbage-collectable is not kept alive just by being used in a mapping.

A typical example is a class that keeps track of its instances, but does not keep them alive just in order to keep track of them:

```
import weakref
class Tracking(object):
    _instances_dict = weakref.WeakValueDictionary()
    def __init__(self):
        Tracking._instances_dict[id(self)] = self
    @classmethod
    def instances(cls): return cls._instances_dict.values
()
```